CRISTINA DIAS URURAHY

AGENTES LUA: UM MECANISMO DE COMUNICAÇÃO EM LUA

DISSERTAÇÃO DE MESTRADO

DEPARTAMENTO DE INFORMÁTICA

Rio de Janeiro, 25 de setembro de 1998

Cristina Dias Ururahy

Agentes Lua: Um Mecanismo de Comunicação em Lua

Dissertação apresentada ao Departamento de Informática da PUC-Rio como parte dos requisitos para a obtenção do título de Mestre em Informática: Ciência da Computação.

Orientadora: Noemi de La Rocque Rodriguez

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro

Rio de Janeiro, 25 de setembro de 1998

Para meus queridos pais

Agradecimentos

Ao Renato por todo seu amor, carinho e dedicação.

À minha família pela educação e apoio que sempre recebi.

Aos meus sogros por seu carinho e entusiasmo.

À Noemi, não só pelo seu incentivo e conhecimento, mas principalmente por sua amizade.

À Dani por ser uma grande amiga e por suas inúmeras caronas.

Ao Borges, Costa e Cassino pela paciência e ajuda durante este trabalho; e ao Roberto por suas contribuições.

A todo o pessoal do LMF, em especial ao Armando, por seus conselhos e sua amizade.

À CAPES pelo auxílio financeiro.

Resumo

O objetivo deste trabalho é avaliar a proposta de usar a linguagem Lua [IFC96] como uma ferramenta para programação paralela em ambientes distribuídos.

O mecanismo utilizado adota uma abordagem orientada a eventos, o que simplifica muitos aspectos de programação concorrente, uma vez que os processos nunca ficam bloqueados em primitivas de comunicação.

A dissertação procura avaliar este mecanismo através de implementações de problemas já codificados com outros mecanismos.

Para estudar estes problemas em Lua, foi feita uma implementação de uma camada sobre o mecanismo original, de forma a facilitar, além de tornar mais confiável, a configuração dos processos e a comunicação entre eles. O trabalho descreve tanto o mecanismo original como esta nova camada.

Também apresentamos as aplicações utilizadas para a avaliação do mecanismo e os resultados dessa avaliação.

Abstract

The object of this work is to evaluate the use of the Lua language [IFC96] as a tool for parallel programming in distributed environments.

The communication mechanism adopts an event-driven approach, that simplifies many aspects of concurrent programming, since processes never block in communication primitives.

The goal of this work is to evaluate this mechanism through implementations of problems previously codified with other mechanisms.

To study these problems in Lua, a layer over the original mechanism has been implemented, in order to make the processes configuration and the communication among them easier and more reliable. This work describes not only the original mechanism, but also this new layer.

We also present the applications that are used for the mechanism evaluation and the results of this evaluation.

Sumário

1 Introdução **1**

2 Mecanismo de Comunicação **4**

2.1 Sistemas Assíncronos 4
2.2 Lua 7
2.3 Agentes Lua 10
2.3.1 Exemplos 11
2.3.2 Implementação 15
2.4 Comunicação 17
2.4.1 Implementação 21
2.4.2 Extensão para C 25

3 Problemas Estudados **29**

3.1 Problema dos N-Corpos 30
3.1.1 Descrição da Aplicação 31
3.1.2 Implementação com os *Agentes Lua* 33
3.2 Visualização Volumétrica Distribuída 36
3.2.1 Descrição do Problema 37
3.2.2 Implementação com os *Agentes Lua* 38

4 Resultados Experimentais **44**

4.1 Resultados do Problema dos N-Corpos 44
4.2 Resultados do Problema da Visualização Volumétrica Distribuída 48
4.3 Observações 52

5 Considerações Finais **53**

A Problemas Estudados **55**

A.1 Problema dos N-Corpos 55
A.1.1 Trecho de Código do Agente Mestre 55
A.1.2 Trecho de Código do Agente Simulador. 56

Lista de Figuras

2.1 Intervalo de tempo de ativação finito. 6
2.2 Intervalo de tempo de ativação instantâneo. 6
2.3 Mecanismo de Comunicação em Lua. 10
2.4 *Hello World* 11
2.5 *Pipeline* com nmáx nós. 12
2.6 *Pipeline* 12
2.7 Mensagem com evento de resposta 13
2.8 Mensagem com evento de Resposta 13
2.9 Exemplo que bloqueia a aplicação 14
2.10 Mensagem quase replicante 15
2.11 Mensagem replicante 16
2.12 Estrutura de um Agente. 17
2.13 Envio e recebimento de mensagens. 18
2.14 Mecanismo Estendido de Comunicação em Lua 19
2.15 *com_spawn* dispara novos agentes 19
2.16 Exemplo do uso da função *com_spawn*. 20
2.17 Variáveis de Configuração 21
2.18 Exemplo de colocação de comandos no *buffer* de envio. 21
2.19 Exemplo de colocar um comando no *buffer* de envio. 21
2.20 Enviando um trecho de código Lua 22
2.21 Exemplo: Multiplicação de Matrizes 23
2.22 Código do exemplo de Multiplicação de Matrizes 24
2.23 Resultado da Multiplicação de Matrizes 25
2.24 Código C do Agente A. 26
2.25 Código C do Agente B. 27
2.26 Transmissão dos dados como strings. 27

3.1 Pseudocódigo do mestre 32
3.2 Pseudocódigo dos Processos Simuladores 33
3.3 Distribuição das Partículas (ORB) 33
3.4 Árvores de Partículas Locais 34
3.5 Árvore de Partículas 34
3.6 Domínio dos Nós 35
3.7 Sincronismo entre os agentes. 36

3.8 Comunicação entre os Agentes. 37
3.9 Estrutura da Aplicação. 38
3.10 Particionamento estático. 38
3.11 Particionamento dinâmico por *scanlines*. 39
3.12 Particionamento dinâmico por blocos. 39
3.13 Algoritmos dos Agentes Visualizador, Particionador e Calculadores. 40
3.14 Trecho de Código do Agente Visualizador. 41
3.15 Trecho de Código do Agente Particionador. 42
3.16 Trecho de Código dos Agentes Calculadores. 43

4.1 Tempo obtido para 1 iteração com 2 nós processadores. 46
4.2 Tempo obtido para 20 iterações com 2 nós processadores. 47
4.3 Tempo obtido para 1 iteração com 4 nós processadores. 47
4.4 Tempo obtido para 20 iterações com 4 nós processadores. 48
4.5 Tempo obtido para 1 iteração com 2.048 partículas. 48
4.6 Tempo obtido para 20 iterações com 2.048 partículas. 49
4.7 Imagem *brain*. 49
4.8 Tempos obtidos para o particionamento dinâmico por blocos. 50
4.9 Tempos obtidos para o particionamento estático por blocos. 51
4.10 Tempos obtidos para o particionamento dinâmico por *scanlines*. 51
4.11 Tempos obtidos para o particionamento estático por *scanlines*. 51

Lista de Tabelas

3.1 Troca das árvores de partículas entre os diversos agentes. 35

4.1 Média da Execução de uma iteração (em ms). 45
4.2 Média da Execução de dez iterações (em ms). 46
4.3 Média da Execução de vinte iterações (em ms). 46
4.4 Tempos obtidos para o particionamento por blocos (em s). 50
4.5 Tempos obtidos para o particionamento por *scanlines* (em s). 50

Capítulo 1

Introdução

Com a proliferação das redes de computadores e a sofisticação dos ambientes multitarefas, o uso de aplicações paralelas e distribuídas vem crescendo rapidamente nos últimos anos. Entretanto, o desenvolvimento de uma aplicação deste tipo pode representar uma tarefa árdua para o desenvolvedor. Aplicações paralelas e distribuídas apresentam, além dos problemas típicos das aplicações seqüenciais, problemas específicos de sua modalidade, tais como instanciação e inicialização de processos, sincronismo, empacotamento e transmissão de dados, tolerância a falhas e, muitas vezes, requisitos rígidos de desempenho.

Muitas vezes, os recursos oferecidos pelas ferramentas tradicionais de desenvolvimento exigem que o desenvolvedor trabalhe com primitivas que têm um nível de abstração muito baixo, dificultando o desenvolvimento destas aplicações.

Para dar um melhor suporte ao desenvolvimento destas aplicações, diversas linguagens paralelas surgiram para oferecer soluções para os problemas de paralelismo e distribuição. Como dificilmente uma linguagem consegue abordar todos estes problemas da forma mais eficaz, algumas linguagens dão um enfoque maior a determinados problemas, tais como sincronismo ([AO93, Bal90]), abstração de dados ([AO93, CK93]) ou desempenho ([CG89, Lov93]).

As linguagens paralelas podem ser classificadas, de acordo com o seu paradigma de comunicação, como logicamente distribuídas, onde os processos comunicam-se através de troca de mensagens, ou como logicamente compartilhadas, onde os processos comunicam-se via memória compartilhada [BST89].

Linguagens logicamente distribuídas podem implementar mecanismos para troca de mensagens síncronas e assíncronas, *rendezvous*, chamada remota de procedimentos, transações atômicas, entre outras. Já linguagens logicamente compartilhadas podem implementar mecanismos para comunicação implícita entre os processos, variáveis compartilhadas ou ainda estruturas de dados distribuídas.

Um aspecto bastante interessante para aplicações distribuídas é a flexibilidade que uma linguagem interpretada pode oferecer. Linguagens interpretadas têm características típicas, tais como interatividade, possibilidade de programação pelo usuário final e reflexividade. Estas características oferecem uma enorme flexibilidade a aplicações finais, como por exemplo, permitir, através de uma console interativa, que o usuário execute comandos de forma

imediata e ainda, tenha acesso direto a todos os recursos do sistema, sem a necessidade de ferramentas específicas. Além disto, o uso de uma linguagem interpretada em sistemas distribuídos pode oferecer uma flexibilidade extra, que é a possibilidade de *migração* de aplicações (como agentes móveis [Gra95]).

Disponibilizando-se todas estas facilidades em uma aplicação distribuída, é possível alterar, em tempo de execução, o comportamento dos *nós* processadores que compõem a aplicação, possibilitando sua configuração dinâmica.

Por outro lado, muitos dos problemas de aplicações distribuídas requerem que as aplicações sejam eficientes e que utilizem toda a potência computacional. Portanto, torna-se de grande interesse ter mecanismos de programação distribuída que ofereçam ao mesmo tempo flexibilidade e eficiência, ou ainda um meio termo entre estas importantes características.

O objetivo deste trabalho é avaliar a proposta de usar a linguagem interpretada Lua [IFC95, IFC96] como uma ferramenta para programação paralela em ambientes distribuídos. Alguns estudos já foram feitos neste sentido, quando proposto um mecanismo básico que estende a linguagem Lua para comunicação entre processos orientada a eventos [CRI95], visando oferecer um simples, no entanto poderoso, mecanismo de comunicação.

O modelo orientado a eventos deste mecanismo se assemelha ao modelo adotado por linguagens logicamente distribuídas que oferecem troca de mensagens assíncronas. Além disso, este mecanismo trata mensagens como trechos de código Lua, que podem ser executados pelo processo receptor. Este comportamento permite que o remetente altere variáveis globais, chame funções e até mesmo defina novas funções no ambiente do receptor, oferecendo assim uma enorme flexibilidade à aplicação.

Do modelo orientado a eventos derivam duas propriedades importantes deste mecanismo. A primeira é que ele é não bloqueante. As primitivas de comunicação são assíncronas e, uma vez que a recepção de mensagens é feita através de um mecanismo dirigido a eventos, um processo nunca fica bloqueado nestas primitivas.

A outra propriedade importante é que cada evento é tratado até ser terminado, antes que o sistema comece a tratar o próximo evento. Esta propriedade é importante pois permite que cada mensagem recebida (bloco de código a ser executado) possa ser considerada um bloco atômico, o que simplifica bastante a aplicação, uma vez que não há concorrência no mecanismo.

Este conjunto de propriedades simplifica muitos aspectos de programação paralela e distribuída, o que pode ser interessante para a prototipação de uma aplicação, além de caracterizar um mecanismo significativamente diferente do modelo cliente/servidor tradicional.

Com base neste mecanismo, já foram estudados problemas típicos[1], porém pequenos, de programação distribuída, com o objetivo de analisar o poder de expressão deste mecanismo. Com este estudo pôde-se observar que o mecanismo oferece grande flexibilidade a aplicações distribuídas [RUIC96a, RUIC96b].

Após este estudo, surgiu a idéia de avaliar este mecanismo em aplicações de maior

[1]Foram estudados algoritmos de detecção de terminação, *probe echo*, *heart-beat* e outros.

porte e considerando também um outro ponto especialmente importante para aplicações paralelas distribuídas: a eficiência. Neste trabalho, procura-se avaliar o custo de usar este mecanismo de comunicação em Lua, isto é, medir qual o preço que se paga por esta flexibilidade. É claro que se espera alguma perda de desempenho, pois afinal, estamos usando uma linguagem interpretada. Porém, quisemos saber qual a ordem de grandeza desta perda.

Para este estudo de aplicações de grande porte, foi feita uma análise de desempenho para duas aplicações. O mecanismo de comunicação destas aplicações, que já estavam implementadas em PVM [Sun90], um mecanismo de comunicação considerado bastante eficiente [MRR96], foi substituído para utilizar os Agentes Lua.

No próximo capítulo, é apresentado inicialmente um resumo sobre as principais características de sistemas assíncronos. Em seguida, é feita uma breve descrição da linguagem Lua [IFC96] e do mecanismo proposto originalmente, concluindo com a descrição da implementação de uma camada sobre o mecanismo de comunicação original. Esta camada tem como objetivo facilitar, além de tornar mais confiável, a configuração dos processos e a comunicação entre eles. No capítulo 3, são apresentados os problemas estudados para a avaliação do mecanismo. São eles o problema dos N-Corpos [FG93] e o da Visualização Volumétrica Distribuída [dBS97], que foram implementados com o mecanismo proposto e em PVM. No capítulo 4, são relatados os resultados de desempenho obtidos através dos problemas estudados. Em seguida, é feita uma análise de desempenho do mecanismo, através de comparações entre os resultados obtidos. Finalmente, no capítulo 5, são apresentadas algumas considerações sobre o que foi apresentado nos capítulos anteriores, assim como alguns trabalhos futuros.

Capítulo 2

Mecanismo de Comunicação

Este capítulo apresenta *Agentes Lua* [CRI95], um mecanismo de comunicação entre processos, que estende a linguagem Lua, através de uma biblioteca de funções para envio e recebimento de mensagens, para programação paralela distribuída.

Neste capítulo, é apresentado inicialmente um resumo sobre as principais características de sistemas assíncronos. Em seguida, é feita uma breve descrição da linguagem Lua [IFC96] e do mecanismo proposto originalmente, concluindo com a descrição da implementação de uma camada sobre o mecanismo de comunicação original.

2.1 Sistemas Assíncronos

Comunicação requer dois processos, um para enviar uma mensagem e outro para recebê-la. Quando um processo envia uma mensagem, é necessário decidir se o receptor deverá cooperar, ou seja, se o receptor deve estar pronto para receber a mensagem quando ela é enviada, ou se o processo pode enviar uma ou mais mensagens desprezando o estado do processo receptor.

Na *comunicação síncrona* a troca de mensagens é uma ação atômica, que requer a participação de ambos os processos: o transmissor – *sender* – e o receptor – *receiver*. Se o transmissor está pronto para enviar a mensagem, mas o receptor não está pronto para recebê-la, o transmissor fica bloqueado e, da mesma forma, se o receptor é o primeiro processo a ficar pronto para a comunicação, ele bloqueia. O ato da comunicação sincroniza as seqüências de execução dos dois processos. O primeiro processo a chegar deve esperar pelo segundo.

Alternativamente, o transmissor pode enviar a mensagem e continuar sem ficar bloqueado. Esta comunicação é chamada *assíncrona*, pois não há uma relação determinada entre as seqüências de execução dos dois processos. O receptor poderia estar executando quaisquer instruções quando a mensagem é enviada e depois, em algum momento mais tarde, verificar se há mensagens no canal de comunicação.

Uma diferença importante entre estes dois modelos de comunicação é a necessidade de armazenar mensagens em um buffer. Na comunicação assícrona, o transmissor pode enviar

muitas mensagens sem que o receptor as remova de seu canal. Portanto o canal deve estar preparado para armazenar um número potencialmente ilimitado de mensagens. Se o número de mensagens no canal é limitado, eventualmente o transmissor será bloqueado. Na comunicação síncrona, apenas uma mensagem existe em qualquer instante de tempo no canal, portanto, não é necessário armazenamento de mensagens.

A diferença é análoga à diferença entre o sistema de telefone e o sistema postal. Uma chamada telefônica sincroniza as ações da pessoa que está ligando e da pessoa que responde. Dificuldades com a sincronização causam sinais de ocupado, chamadas não atendidas, etc. Por outro lado, qualquer número de cartas pode ser deixado numa caixa de correio a qualquer momento e o receptor pode escolher verificar sua caixa de correio a qualquer instante. Não há sincronização.

No modelo assíncrono, a recepção de mensagens pode ocorrer segundo o modelo orientado a eventos. Ou seja, o receptor não verifica se uma mensagem chegou através de uma primitiva de `receive`, mas através de um *loop* de eventos que consulta que eventos a aplicação está recebendo. Além de eventos de rede, a aplicação pode recever eventos de interface com o usuário, entre outros.

Em Ada [Ada79], assim como nos Agentes Lua, as mensagens são recebidas de forma não determinística. Porém, apesar do não determinismo de Ada, ao receber uma mensagem, os nós (também chamados *tasks*) têm um leque de possíveis mensagens que podem ser recebidas, enquanto nos Agentes Lua qualquer tipo de mensagem pode ser recebida.

Isto deve-se ao fato de que, em Ada, o programador pode usar a primitiva de `select` no receptor para dizer que tipo de mensagem pode atingir aquele nó.

Nos Agentes Lua esta facilidade não existe explicitamente, apesar do usuário poder implementar um mecanismo similar.

O modelo utilizado nos Agentes Lua é um modelo de comunicação assíncrona, que se assemelha em alguns aspectos ao modelo atômico de computação distribuída [Mat87]. Neste modelo, um sistema distribuído consiste de um conjunto fixo de processos que se comunicam exclusivamente através de mensagens. Todas as mensagens são recebidas corretamente após um *delay* arbitrário, mas finito, e a comunicação é assíncrona, ou seja, um processo nunca espera que o receptor fique pronto antes de enviar a mensagem. Portanto, assume-se que o sistema de comunicação tenha uma capacidade ilimitada de armazenamento de mensagens (ou, tomando uma visão mais realista, que mantenha um mecanismo de controle de fluxo transparente). Não é necessário que mensagens enviadas através do mesmo canal de comunicação obedeçam a regra *FIFO – first in, first out*.

O modelo atômico de computação distribuída apresenta as seguintes características:

1. Em qualquer momento, um processo está ou ativo ou inativo.

2. Apenas processos inativos podem receber mensagens. [1]

[1]Esta restrição não é muito séria, uma vez que devido a (5) um processo pode mudar de ativo para inativo um pouco antes de receber uma mensagem. Esta condição também pode ser cumprida armazenando-se as mensagens recebidas em um buffer e considerando-se este buffer de mensagens como parte lógica do sistema de comunicação.

3. No recebimento de uma mensagem um processo pode passar de inativo para ativo.

4. Apenas processos ativos podem enviar mensagens. [2]

5. Um processo pode mudar de ativo para inativo em qualquer instante.

Estamos considerando apenas sistemas nos quais todos os processos irão eventualmente se tornar inativos, embora esta propriedade seja geralmente indecidível.

Se um processo só está ativo durante um intervalo de tempo finito $[t1, t2]$, a duração exata de sua fase ativa é irrelevante. Uma vez que o atraso de uma mensagem é arbitrário, todas as mensagens enviadas neste intervalo de tempo poderiam ter sido enviadas no início dessa fase, levando um pouco mais de tempo em seu caminho para o processo destino (figuras 2.1 e 2.2).

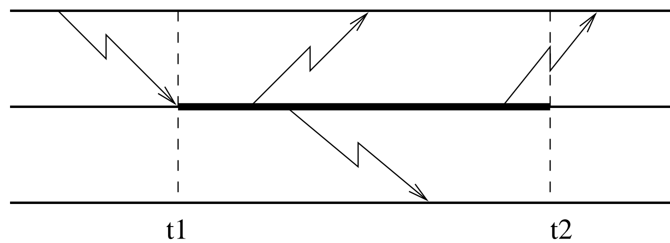


Figura 2.1: Intervalo de tempo de ativação finito.

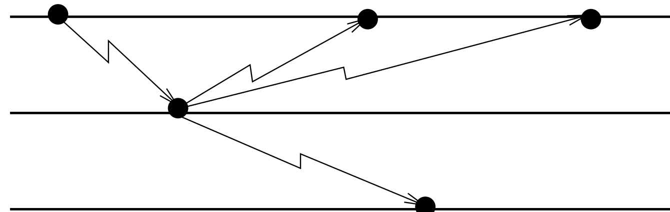


Figura 2.2: Intervalo de tempo de ativação instantâneo.

No modelo atômico da computação distribuída, um processo pode em qualquer momento pegar qualquer mensagem de um de seus canais de comunicação de entrada (desde que exista alguma), imediatamente mudar seu estado interno e no mesmo instante enviar um número qualquer de mensagens, possivelmente nenhuma (de modo que a computação possa terminar). Para ter uma visão mais vívida do modelo atômico, mensagens podem ser pensadas como fluindo constantemente porém com velocidades variadas para seu destino, finalmente atingindo um processo. Então ou nada acontece e a mensagem é silenciosamente consumida, ou novas partículas são ejetadas, como ocorre em uma reação atômica.

[2] Uma vez que não estamos preocupados com o problema de inicialização, assumimos que todos os processos estão inicialmente inativos e uma mensagem chega de fora do sistema para iniciar a computação; alternativamente podemos assumir que inicialmente um processo pode estar ou ativo ou inativo.

Outra característica importante do modelo atômico de computação distribuída é que toda computação local, iniciada pelo receptor da mensagem, deve terminar. O modelo atômico, que é semelhante ao modelo de *atores* [Cli81], é especialmente apropriado quando utilizado sobre sistemas distribuídos, porque não há atividade concorrente em um processo. Isto contrasta com o modelo CSP síncrono [Hoa85], no qual o envio de mensagens é instantâneo, mas processos estão ativos concorrentemente.

2.2 Lua

Lua é uma linguagem projetada especificamente para ser usada como uma linguagem de extensão [IFC95, IFC96]. Uma linguagem de extensão é normalmente utilizada em conjunto com uma outra linguagem, como C ou Modula-2. Neste modelo, os programas são divididos em duas partes, *núcleo* e *configuração*. O núcleo implementa os objetos e classes básicas do sistema, e é normalmente escrito em uma linguagem compilada e estaticamente tipada. A parte de configuração, normalmente escrita em uma liguagem interpretada e flexível, conecta estas classes e objetos para dar uma forma final à aplicação.

Em uma aplicação deste tipo, a parte do núcleo interage com o interpretador Lua através de uma API que fornece funções para realizar diversos tipos de serviços, tais como executar pedaços de código Lua e manipular variáveis globais de Lua.

Lua é uma linguagem interpretada, tem uma sintaxe simples e uma semântica simples, é pequena, portátil, além de ser bastante flexível.

Entre os mecanismos que tornam Lua uma linguagem bastante flexível, podemos destacar: *arrays* associativos, que agem como construtores de dados; recursos de reflexividade, que permite facilidades para *self manipulation*; e a facilidade de incorporação de bibliotecas C, que permitem adicionar novas funcionalidades à linguagem.

A unidade de execução de Lua é chamada um *trecho de código Lua*. Um trecho de código Lua pode conter comandos e definições de funções, e pode ser um arquivo ou uma cadeia de caracteres.

Quanto ao seu sistema de tipos, Lua é uma linguagem dinamicamente tipada. Variáveis não possuem tipos; somente valores têm tipo. Existem seis tipos básicos em Lua: valor nulo (`nil`), números de ponto flutuante (`number`), cadeias de caracteres (`string`), funções (`function`), dado do usuário (`userdata`) e tabelas (`tables`).

Entre as características mais relevantes da linguagem para este trabalho, estão

- **o ambiente global de Lua**. Todos os comandos avaliados pelo interpretador são executados no ambiente global de cada agente. Além disso todas as modificações que um trecho efetua sobre o ambiente global persistem após o seu término.
- **a função *dostring***, que recebe uma *string* como parâmetro, que é, na realidade, um trecho de código Lua. Este código é executado pelo interpretador Lua, no ambiente global.
- **o uso de *strings***. *Strings* podem ser representadas de três formas distintas, como no exemplo abaixo:

```
print( "Hello World" )
print( 'Hello World' )
print([[Hello World]])
```

A diferença entre elas é que apenas a última permite que haja aninhamento, com o mesmo delimitador. Observe os exemplos abaixo.

```
cmd = "print( "Hello World" )"  -- Incorreto.
cmd = 'print( 'Hello World' )'  -- Incorreto.
cmd =[[print([[Hello World]])]] -- Correto.
cmd = 'print( "Hello World" )'  -- Correto.
cmd = "print( 'Hello World' )"  -- Correto.
```

- **concatenação de *strings***. Para fazer a concatenação de *strings* existe o operador "..". Considere o exemplo a seguir.

```
i = 1
while i <= 3 do
   print("Iteração ".. i)
   i = i + 1
end
```

O resultado da execução deste trecho será :

```
Iteração 1
Iteração 2
Iteração 3
```

- **o uso de tabelas**. O tipo `table` implementa *arrays* associativos, isto é, *arrays* que podem ser indexados por qualquer tipo de valor da linguagem. Este tipo pode ser usado não somente para representar *arrays* convencionais, como também tabelas de símbolos, conjuntos, estruturas (*records*), etc.

  É importante destacar que tabelas são objetos e não valores. Variáveis não podem conter tabelas, somente referências para elas. Atribuição, passagem de parâmetro e retorno de função sempre manipulam referências para tabelas, e não implicam em nenhum tipo de cópia. Além do mais, tabelas têm que ser explicitamente criadas antes de serem usadas.

  Os exemplos a seguir mostram algumas maneiras de criar tabelas.

```
t1 = { 1, 2, 3, 4, 5}   é equivalente a     t1 = {}
                                            t1[1] = 1
                                            t1[2] = 2
                                            t1[3] = 3
                                            t1[4] = 4
                                            t1[5] = 5

t2 = {"a", "b", "c"}    é equivalente a     t2 = {}
                                            t2[1] = "a"
                                            t2[2] = "b"
                                            t2[3] = "c"

t3 = { 1, "a"; x=10}    é equivalente a     t3 = {}
                                            t3[1] = 1
                                            t3[2] = "a"
                                            t3["x"] = 10 ou t3.x = 10
```

- **definição de funções.** Funções são tratadas como qualquer outro tipo de valor. Assim, funções podem ser atribuídas a variáveis, passadas como argumento para outras funções ou retornadas como resultado. Quando uma função é definida em Lua, seu corpo é compilado e armazenado em uma determinada variável.

  Com relação a funções, cabe destacar que a passagem de parâmetros é feita por valor, e funções podem receber um número variável de parâmetros e retornar mais de um valor. Isto torna desnecessária a passagem de parâmetro por referência quando mais de um resultado precisa ser retornado por uma função.

  Observe os exemplos abaixo.

```
function soma(a, b)
   return a+b
end

add = soma

print(add(3, 5))          -- o resultado será 8.


function divide( dividendo, divisor)
   quociente = floor( dividendo/divisor)
   resto     = mod( dividendo, divisor)
   return quociente, resto
end

q, r = divide( 12, 5)
```

- **registro de uma função C**. Para que uma função C possa ser utilizada em um programa Lua é necessário que esta função seja **registrada** em Lua. Para isto, o programa C deve utilizar a função `lua_register`.

  Considere o exemplo a seguir.

```
/* Código C */

void soma_C()
{
float a = lua_getnumber( lua_getparam(1));
float b = lua_getnumber( lua_getparam(2));

lua_pushnumber( a+b);
}

void main()
{
...
lua_register("soma", soma_C);
...
}


-- Código Lua

c = soma(5, 6)            -- O valor de c será 11.
```

2.3 Agentes Lua

Agentes Lua [CRI95] é uma extensão da linguagem Lua, que implementa comunicação baseada em eventos. O mecanismo dos Agentes Lua baseia-se no modelo de comunicação assícrona, apresentado na seção 2.1. O modelo dos Agentes Lua também é bastante parecido com o utilizado em [Bar96] para descrição de algoritmos distribuídos.

Neste modelo, cada agente é um processo independente que se comunica com os demais agentes através de mensagens assíncronas. Cada agente possui um interpretador Lua. Além do interpretador, cada processo possui um *loop* de eventos de forma a poder gerenciar todos os eventos de rede e de interface com o usuário. Através de sua interface, o usuário pode fornecer trechos de código Lua que serão interpretados pelo agente. Este trechos podem incluir a execução de um arquivo (através do comando *dofile*), ou ainda o envio de mensagens para outros agentes. Da mesma forma, todos os eventos de rede recebidos são tratados como trechos de código Lua e são executados pelo interpretador de cada agente.

A função *send* tem um comportamento convencional, enviando uma *string* assincronamente para um receptor específico. O sistema não possui uma função simétrica para receber mensagens. Ao invés disso, as mensagens são recebidas como eventos especiais (figura 2.3).

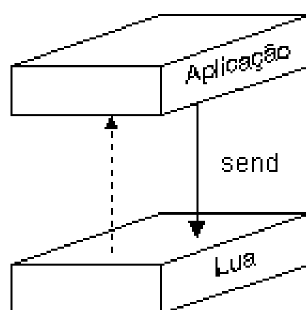


Figura 2.3: Mecanismo de Comunicação em Lua.

As mensagens enviadas são, na realidade, trechos de código Lua que serão executados pelo agente receptor em seu ambiente global.

Outra característica importante do mecanismo é que ele trata cada uma das mensagens recebidas como um trecho de código atômico, ou seja, cada evento (trecho de código a ser executado) é tratado até ser terminado, antes que o sistema comece a tratar o próximo evento.

Um mecanismo de comunicação entre aplicações através de uma linguagem de extensão baseado em uma função do estilo de `send` foi proposto anteriormente para a linguagem de extensão TCL [Ous90]. Neste mecanismo, a mensagem também é encarada como um trecho de código a ser executado pelo processo receptor. No entanto, esse mecanismo só trata a comunicação entre processos em uma mesma máquina, sem se preocupar com o

aspecto de programação distribuída.

Outro mecanismo similar ao dos Agentes Lua é apresentado em [Bic95]. Nesse trabalho é apresentado o sistema MESSENGERS que se baseia em um paradigma chamado *Objetos Autônomos*. No modelo de Objetos Autônomos, o processamento realizado em cada nó de um sistema distribuído é uma seqüência de instruções executando assincronamente e operando sobre dados locais não compartilhados. Os Agentes Lua se adequam muito bem a este paradigma, especialmente em aplicações tais como a descrita em [RUIC96a].

O sistema MESSENGERS é baseado na idéia genérica de objetos inerentemente autônomos com ênfase em composição e coordenação de atividades concorrentes em um ambiente distribuído. Enquanto MESSENGERS têm comandos explícitos para computação, navegação e interação, os Agentes Lua não oferecem nenhuma facilidade explícita para migração de código [Gra95]. Entretanto, o fato das mensagens poderem carregar código executável permite que aplicações migratórias sejam implementadas sobre os Agentes Lua (ver Mensagem Replicante na seção 2.3.1).

2.3.1 Exemplos

Nos Agentes Lua, a primitiva de `Receive` em Lua executa a mensagem recebida. Porém, não é o programador quem decide a hora de receber uma mensagem, mas sim o sistema. Portanto, a função `Receive` é chamada internamente pelo sistema quando há um evento de rede a ser tratado. Note que o recebimento de uma mensagem não fica explícito no código de um programa, mas ocorre em função do evento de chegada de uma mensagem. Para ilustrar este mecanismo considere os exemplos a seguir.

"Hello World"

Considere que existem dois agentes (AgenteA e AgenteB). Cada um deles possui a variável `hello` com um conteúdo diferente. O AgenteA enviará uma mensagem ao AgenteB, pedindo que ele mostre o conteúdo da variável `hello` definida no agente receptor. Em seguida, outra mensagem será enviada pelo AgenteA, pedindo que ele mostre o conteúdo da variável `hello` definida no agente que enviou a mensagem. Este trecho de código pode ser observado na figura 2.4.

```
hello = "Hello World A"

send("AgenteB", "print(hello)")

send("AgenteB", "print('".. hello ..."')")
```

(2.4a) AgenteA

```
hello = "Hello World B"
```

(2.4b) AgenteB

Figura 2.4: *Hello World*

Note que no primeiro *send*, a mensagem executada no agente receptor (AgenteB) será `print(hello)`, e portanto, será exibido o texto *Hello World B*. Já no segundo *send*, o uso

da concatenação, que é feita no AgenteA, vai gerar a mensagem `print('Hello World A')` que será enviada ao agente receptor. A segunda mensagem fará com que seja exibido o texto *Hello World A*.

Pipeline

Considere o exemplo de enviar mensagens através de um *pipeline* de *nmax* nós(figura 2.5). Cada nó deste *pipeline* está ligado aos outros nós através de dois canais de comunicação. Um, representado à sua esquerda, por onde ele recebe mensagens, e outro, à sua direita, por onde ele envia mensagens. O objetivo é fazer com que o primeiro nó do *pipeline*, que contém *nmsg* mensagens, envie estas mensagens para os demais nós.[3]

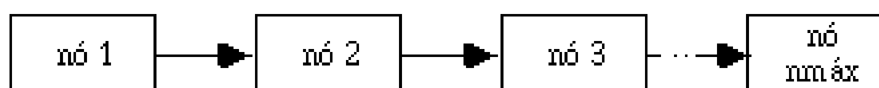


Figura 2.5: *Pipeline* com nmáx nós.

Na figura 2.6 pode ser observado o código deste exemplo, que é o mesmo para cada um dos nós, em uma linguagem com as primitivas de *send* e *receive* e na linguagem Lua.

```
...
for k := 1 to nmsg do
    begin
       if nó > 1 then
          receive(left, b[k]);
       end;
       if nó < nmax then
          send(right, b[k]);
       end;
    end;
```

(2.6a) Linguagem com *send* / *receive*

```
function foo(msg)
   b[k] = msg
   if nó < nmax then
      send(right, "foo("..b[k]..")")
   end
   k = k + 1
end

...
k = 1
if nó == 1 then
   while k <= nmsg do
      send(right, "foo("..b[k]..")")
      k = k + 1
   end
end
```

(2.6b) Lua

Figura 2.6: *Pipeline*

No exemplo 2.6a, todos os nós (menos o primeiro) ficam bloqueados na primitiva de *receive* até que chegue uma mensagem pelo seu canal esquerdo. Assim que chega uma mensagem, todos os nós que estavam bloqueados (exceto o último) enviam a mensagem recebida pelo seu canal direito. Desta forma, uma mensagem enviada pelo nó 1 é recebida

[3]Na verdade, nos Agentes Lua não existem apenas estes dois canais de comunicação para cada um dos agentes. Este exemplo apenas modela esta situação. Nos agentes Lua, cada um dos agentes pode receber e enviar mensagens de todos os demais agentes sem que canais de comunicação específicos tenham que ser criados para isso.

pelo nó 2 e é em seguida enviada ao nó 3, e assim sucessivamente até que a mensagem chegue ao último nó (nó nmax), que não a transmitirá.

No exemplo 2.6b, nenhum nó fica bloqueado. Cada mensagem enviada pelo nó 1 é uma chamada à função `foo`, que fará com que o agente receptor da mensagem (exceto o último nó) envie uma mensagem semelhante à recebida ao nó à sua direita. Assim, o nó 1 envia uma mensagem ao nó 2. Ao receber esta mensagem, o nó 2 executa a função `foo` e envia uma mensagem semelhante à recebida ao nó 3, e assim sucessivamente, cada vez que uma nova mensagem é recebida.

Mensagem com Evento de Resposta

Um outro exemplo que mostra a flexibilidade deste mecanismo é apresentado nas figuras 2.7 e 2.8. Neste exemplo, o AgenteA deseja saber quanto vale a variável n no AgenteB Para isto ele envia uma mensagem para o AgenteB que é um comando Lua que envia uma mensagem para o AgenteA. No exemplo, o AgenteA deverá exibir o valor de n.

```
-- AgenteA

   msg = "send('AgenteA', 'print('.. n .. ')')"

   send("AgenteB", msg)
```

Figura 2.7: Mensagem com evento de resposta

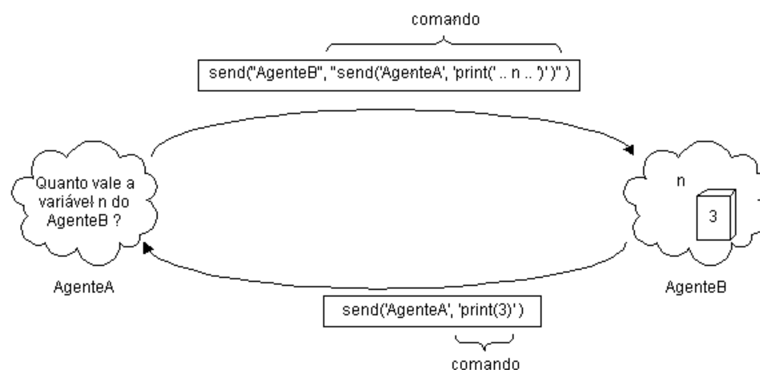


Figura 2.8: Mensagem com evento de Resposta

A mensagem `send('AgenteA', 'print('..  n ..  ')')` será recebida pelo AgenteB. Ao executar a mensagem recebida, o AgenteB enviará ao AgenteA a mensagem `print(3)`. Note que a concatenação foi resolvida no AgenteB e portanto, o valor de `n` que será exibido pelo AgenteA, será o valor de `n` conhecido pelo AgenteB (`n` é uma variável global Lua no ambiente do AgenteB).

Na abordagem orientada a eventos, é necessário que o programador fique atento para algumas características específicas deste paradigma de programação. O programador deve sempre lembrar-se que cada evento é tratado até ser terminado, antes que o sistema comece a tratar o próximo evento.

No exemplo mostrado na figura 2.9, o AgenteA deseja enviar uma mensagem ao AgenteB e ter uma confirmação de que ele a recebeu. O exemplo 2.9a bloquearia a aplicação, pois ela ficaria presa no *loop*, esperando a variável *msg_received* ser sinalizada e não conseguiria tratar a mensagem que sinaliza esta variável. Uma forma de escrever este mesmo exemplo sem bloquear a aplicação é apresentado na figura 2.9b. O AgenteB receberá a mensagem `send('a','Answer()')`. Desta forma, o AgenteA irá executar a função `Answer` quando a mensagem enviada pelo AgenteB for recebida.

```
-- AgenteA

...
   msg_received = nil

   msg = "send('AgenteA','msg_received = 1')"
   send('AgenteB', msg)

   while ~msg_received do
   end
   print("The End.")
```

(2.9a) Bloqueia a aplicação

```
-- AgenteA
   function Answer()
      print("The End.")

   end


   msg = "send('AgenteA','Answer()')"
   send('AgenteB', msg)
```

(2.9b) Não bloqueia a aplicação

Figura 2.9: Exemplo que bloqueia a aplicação

Note que a mensagem enviada pelo AgenteA é uma mensagem com evento de resposta.

Mensagem Quase Replicante

O exemplo a seguir tem o objetivo de enviar uma mensagem que se replica por todos os agentes.

Seguindo o exemplo 2.7, vamos supor agora que o AgenteA deseja saber o valor da variável `n` em cada um dos agentes que estão se comunicando com ele. Neste exemplo cada um dos agentes possui uma variável chamada `nexthost`, que contém o nome do agente que está à sua direita, ligando os agentes como em um *pipeline*.

A figura 2.10 mostra um exemplo onde a mensagem enviada pelo AgenteA atinge apenas os dois primeiros agentes (AgenteB e AgenteC), mas não consegue atingir os demais agentes em questão. Neste exemplo existe um número fixo de envios contidos na mensagem enviada pelo AgenteA. Vamos chamá-la de mensagem *quase replicante*.

Para atingir o objetivo deste exemplo é necessário fazer com que o mesmo trecho de código seja executado em cada um dos agentes. Este trecho de código consiste em enviar uma resposta para o agente a e enviar outro pedido para o agente à sua direita. Neste exemplo, o agente b envia para o agente c uma mensagem diferente da que ele recebeu.

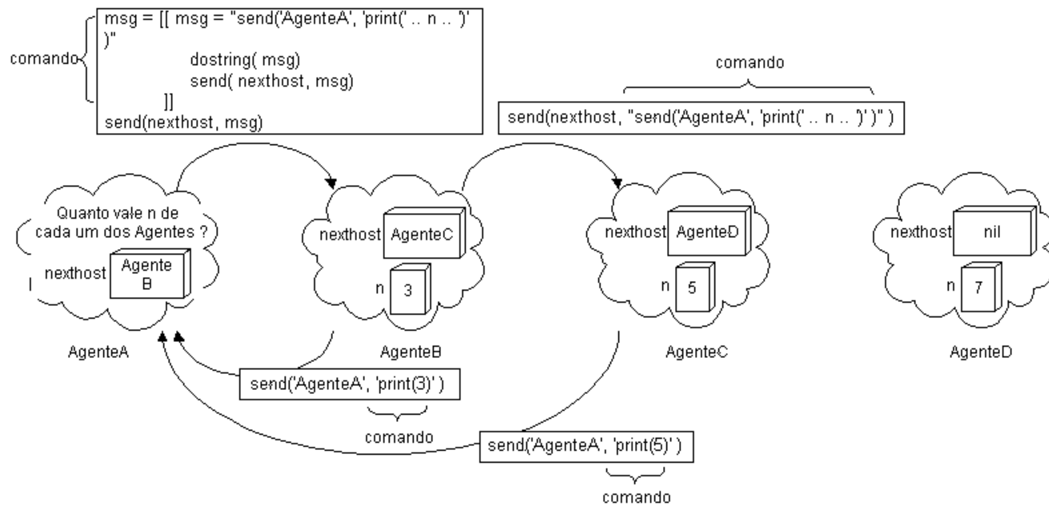


Figura 2.10: Mensagem quase replicante

Mensagem Replicante

Para que o exemplo anterior funcione da forma desejada é necessário fazer com que os agentes enviem a mesma mensagem que foi recebida, como é apresentado na figura 2.11.

Note que a mensagem enviada pelo AgenteA ao AgenteB contém a definição da variável `msg` e um comando que a executa. Quando a mensagem é executada pelo AgenteB, o AgenteB define a variável `msg` e a executa. Ao executá-la, ele envia ao AgenteA o valor de sua variável `n` e envia ao agente à sua direita a definição da variável `msg` e um comando com sua execução [4], que foi exatamente o trecho de código que ele recebeu, e portanto este agente irá enviar a mesma mensagem recebida ao agente a sua direita. Desta forma a mesma mensagem é enviada a todos os agentes e todos eles conseguem enviar ao AgenteA o valor da variável `n`, que era o objetivo inicial deste exemplo.

O exemplo do *pipeline* (figura 2.6) poderia ser reescrito usando este tipo de mensagem. A mensagem enviada teria que ser transmitida até o último nó do *pipeline*, porém, neste caso, não haveria necessidade de haver um evento de resposta.

2.3.2 Implementação

O mecanismo apresentado consiste em disponibilizar em Lua, através de uma biblioteca, apenas uma função para comunicação, chamada *send* e em tratar eventos de rede quando a aplicação estiver ociosa.

A estrutura de um agente Lua consiste de seis elementos (figura 2.12). O *dispatcher*, que gerencia eventos de rede e de interface; uma fila de eventos, que contém as mensagem recebidas que ainda não foram tratadas; uma interface com o usuário, através da qual ele pode fornecer código Lua a ser executado; a biblioteca de funções de comunicação, com

[4] `send( nexthost, "msg=[["..msg.."]]; dostring(msg)")`

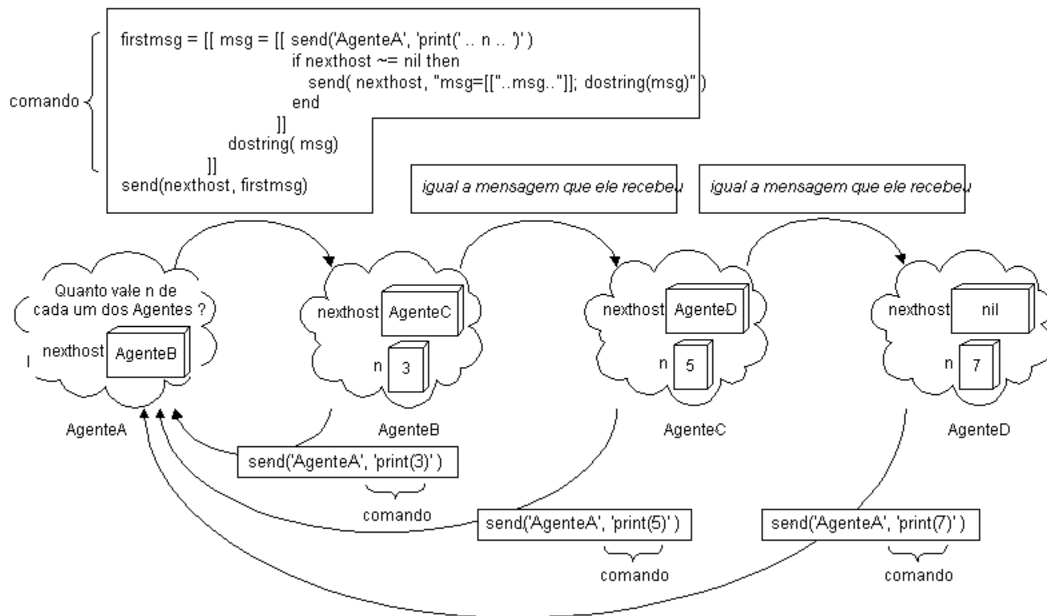


Figura 2.11: Mensagem replicante

funções de enviar mensagens, consultar a fila de eventos, tratar uma mensagem da fila de eventos e outras; o interpretador de código Lua, para que as mensagens recebidas, assim como os comandos fornecidos pelo usuário possam ser executados[5] e uma aplicação, que engloba definição de funções e variáveis, assim como qualquer código Lua, definido pelo usuário.

Através deste mecanismo de gerência de eventos, a aplicação fica em *loop* cuidando dos eventos relacionados à interface com o usuário (que pode incluir a execução de um código Lua) e é definida uma função que trata dos eventos de comunicação, que é chamada apenas quando a aplicação está ociosa. Quando existe um evento de comunicação a ser tratado, é chamada a função `Receive` com a mensagem e o remetente como parâmetros. A implementação pré-definida para esta função é *executar* a mensagem recebida (através do comando *dostring*), assumindo que ela é um trecho de código Lua (ver figura 2.13). Este comportamento oferece uma enorme flexibilidade ao mecanismo de comunicação, permitindo que o remetente altere variáveis globais, chame funções e até mesmo defina novas funções no ambiente do receptor.

Tanto a função `send` quanto a função responsável pelos eventos de comunicação foram implementadas em C, usando o mecanismo de soquetes para envio de datagramas [Ste90, CS93].

A função `Receive`, que trata a mensagem recebida, assume que a *string* da mensagem é um trecho de código Lua, que será executado pelo interpretador Lua do agente receptor do evento.

Cabe destacar que a função `Receive` só é chamada pelo sistema quando este identifica

[5]As mensagens recebidas são tratadas como código Lua.

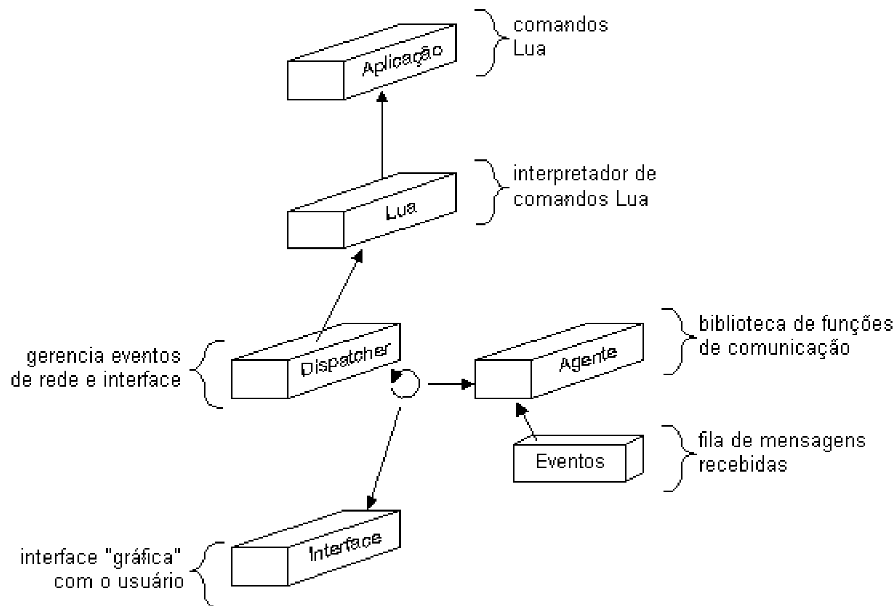


Figura 2.12: Estrutura de um Agente.

um evento de comunicação; a aplicação nunca chama esta função explicitamente.

Outro aspecto a ser destacado é que este código será executado no ambiente global de Lua do agente receptor. Este código pode alterar valores de variáveis e chamar funções já existentes no ambiente do agente, assim como pode definir novas variáveis e funções.

Uma propriedade importante do protocolo acima é que ele é não bloqueante. A função `send` é, como já foi mencionado, assíncrona, e uma vez que a recepção é feita através de um mecanismo dirigido a eventos, um processo nunca fica bloqueado. Outra propriedade importante é que cada evento é tratado até ser terminado, antes que o sistema comece a tratar o próximo evento. Esta propriedade é importante pois permite que cada mensagem recebida (bloco de código a ser executado) possa ser considerada um bloco atômico, ou seja, um bloco que não terá sua execução interrompida.

Nesta implementação dos Agentes Lua não garantimos uma capacidade ilimitada de armazenamento das mensagens recebidas. Neste trabalho, não foram implementados mecanismos de controle de fluxo transparente. Portanto, se a taxa de recebimento de mensagens de um agente for maior que a sua taxa de processamento das mensagens, pode ocorrer perda de mensagens se o *buffer* de recepção estourar.

2.4 Comunicação

[RUIC96a] e [RUIC96b] descrevem um estudo da viabilidade do mecanismo apresentado acima para programação distribuída. Neste trabalho foram estudados problemas típicos, porém pequenos, de programação distribuída. Foram implementados algoritmos de detecção de terminação de uma aplicação, utilizando algoritmos como *probe/echo* e *token*

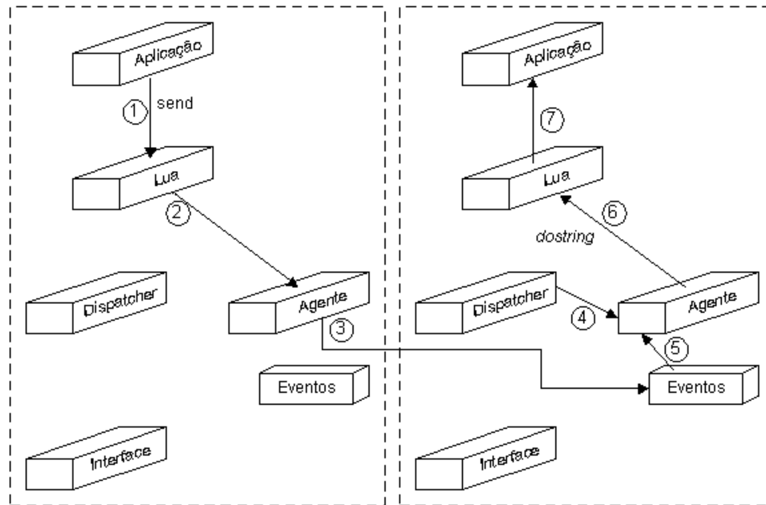


Figura 2.13: Envio e recebimento de mensagens.

ring [Mat87]. O objetivo deste trabalho era analisar o poder de expressão dos Agentes Lua. Para isto, os mesmos algoritmos foram implementados em outras linguagens, como SR [AO93] e C (soquetes) [Ste90]. Com este estudo pôde-se observar que os Agentes Lua oferecem grande flexibilidade a aplicações distribuídas.

Esta seção apresenta um mecanismo de comunicação entre processos para programas Lua, que é uma extensão do mecanismo básico dos Agentes Lua, com funções específicas para facilitar seu uso, além de fazê-lo mais confiável.

Uma vez que o mecanismo original é bastantes simples, com apenas uma função de envio de mensagens, surgiu a necessidade de se criar uma nova camada sobre este mecanismo básico. Esta nova camada oferece alguns recursos adicionais, tais como disparar novos agentes, terminar a execução de um determinado agente, e verificar se um determinado agente está recebendo e enviando mensagens (figura 2.14). Dentre os recursos adicionais oferecidos está um *buffer*, cuja utilização é similar à utlização do *buffer* oferecido pela biblioteca PVM. Com este *buffer*, o usuário pode montar a mensagem a ser enviada no *buffer*, para depois enviá-la a um ou mais agentes.

Entre as funções criadas está *com_spawn* que dispara n agentes para serem usados na aplicação (figuras 2.15 e 2.16). Ao disparar novos agentes, a função *com_spawn_completed* é chamada pelo próprio sistema, assim que os novos agentes estiverem prontos para receber e enviar mensagens. Está função é executada no agente que chamou a função *com_spawn*.

No exemplo da figura 2.16, após disparar dois novos agentes, o agente mestre envia uma mensagem vazia (uma vez que nada foi inserido no *buffer*), com cabeçalho "Acordei" para cada um deles, através do comando *com_mcast*. Assim que as mensagens forem recebidas nos novos agentes, o cabeçalho da mensagem será exibido no agente receptor.

Cada agente possui um conjunto de variáveis de configuração em seu ambiente global, tais como seu *nome de acesso*, o nome do agente que o criou, e o indentificador do agente (figura 2.17).

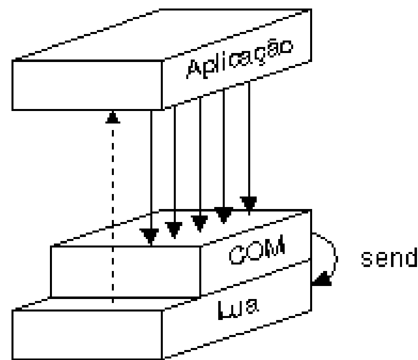


Figura 2.14: Mecanismo Estendido de Comunicação em Lua

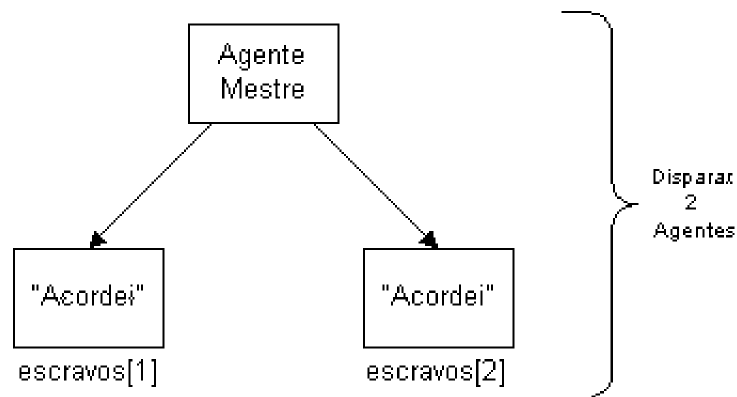


Figura 2.15: *com_spawn* dispara novos agentes

Uma vez que todos os agentes estejam criados, a idéia é montar a mensagem que se deseja enviar em um *buffer*. Após a mensagem desejada estar completa, ela pode ser enviada para um ou mais agentes.

Cabe aqui lembrar que estas mensagens são na verdade comandos Lua, que podem estar na sua forma mais simples, como uma atribuição, ou como um comando de *while* ou ainda a definição de uma função.

Note que o uso deste *buffer* pode ser particularmente interessante no caso de se querer enviar comandos mais complexos a outros agentes, uma vez que a concatenação de *strings* para montar os comandos desejados poderia ser uma tarefa um pouco árdua para o desenvolvedor. Com o uso do *buffer*, os comandos podem ser montados com a utilização de primitivas especialmente criadas para facilitar esta tarefa.

Para exemplificar o uso do *buffer*, considere o exemplo da figura 2.18. Ao colocar uma mensagem no *buffer* de envio, basta lembrar que esta mensagem será executada no processo que a receber, portanto a mensagem (ou comando) deve poder ser executada. Através do comando *com_exc*, um comando é colocado no *buffer* de envio. Quando o *buffer* for enviado, todas as mensagens armazenadas anteriormente serão tratadas como um único comando.

Note ainda que, no exemplo anterior, o bloco de *while* poderia ser colocado no *buffer*

```
function start()
        com_spawn(2)                          -- dispara 2 agentes
end

function com_spawn_completed( escravos)       -- será chamada pelo "mestre" assim que
                                              -- os agentes forem disparados.

        com_initsend()                        -- inicializa a comunicação

        com_mcast( escravos, "Acordei")       -- envia uma mensagem vazia, com
                                              -- cabeçalho "Acordei" para todos os agentes
                                              -- contidos na tabela "escravos"
        print("Acabei...")
end

start()
```

Figura 2.16: Exemplo do uso da função *com_spawn*.

de envio de uma única vez, como mostra a figura 2.19.

Um exemplo que dispara dois agentes e envia o mesmo trecho de código para ser executado pode ser observado na figura 2.20.

Dado que é muito comum querer enviar o conteúdo de uma variável a outros agentes, além da função *com_exc*, existe a função *com_pk* que também coloca um comando no *buffer* de envio. A diferença é que a primeira recebe como parâmetro um código Lua e o coloca no *buffer* sem alterá-lo. Já a segunda recebe dois parâmetros. O segundo é o valor de uma variável, que pode ser um número, uma *string* ou ainda uma referência para uma tabela [6] e o primeiro parâmetro é o nome que esta variável terá no ambiente do agente receptor. A função *com_pk* transforma o conteúdo da variável (segundo parâmetro) em uma string e monta um comando de atribuição com o valor do primeiro parâmetro.

A seguir é apresentado um exemplo de multiplicação de matrizes, que pode ser observado na figura 2.21. As matrizes utilizadas são 2 x 2 e são necessários 4 agentes, onde cada um deles irá multiplicar uma linha por uma coluna.

Primeiramente são disparados os quatro agentes necessários. Quando eles já estão prontos para receber mensagens, é enviada a definição da função que multiplica uma linha por uma coluna, para todos os agentes disparados. Em seguida é enviada para cada um deles uma mensagem com a linha e a coluna que cada agente deverá multiplicar, além da chamada da função de multiplicação que somente será executada no agente receptor. A função de multiplicação envia uma mensagem para o agente *mestre* informado-lhe o resultado de sua tarefa. Este código é apresentado na figura 2.22.

O resultado da execução deste exemplo, através do uso de uma interface gráfica, pode ser observado na figura 2.23.

[6]Note que não faz sentido querer enviar um tipo *userdata* nem uma função já compilada, pois estas não fariam sentido no ambiente do agente receptor.

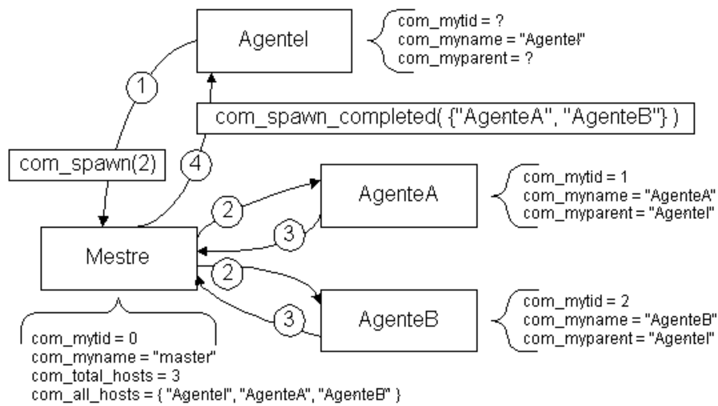


Figura 2.17: Variáveis de Configuração

```
com_exc("a=3")
com_exc("while a > 0 do")
com_exc("   print(a)    ")
com_exc("   a = a - 1   ")
com_exc("end            ")
```

Figura 2.18: Exemplo de colocação de comandos no *buffer* de envio.

2.4.1 Implementação

Além das funções disponíveis para o usuário, outras funções auxiliares foram desenvolvidas.

Para que não haja problemas quando a mensagem a ser enviada for maior que o tamanho que a função *send* consegue enviar, foi implementado um mecanismo de quebra de mensagens. O *buffer* utilizado para montar a mensagem a ser enviada é dividido em vários campos. Quando o usuário pede para colocar uma mensagem no *buffer*, verifica-se se ela ainda cabe no campo do *buffer* corrente. Caso ela ultrapasse o tamanho permitido, um novo campo é criado.

Cada um dos campos do *buffer* é enviado aos processos destino através de uma mensagem. Em cada mensagem, é inserido um código extra para que elas possam ser concate-

```
msg = [[ a=3
         while a > 0 do
            print(a)
            a = a - 1
         end
      ]]
com_exc(msg)
```

Figura 2.19: Exemplo de colocar um comando no *buffer* de envio.

```
function start()
        com_spawn(2)                            -- dispara 2 agentes
end

function com_spawn_completed( escravos)         -- será chamada pelo "mestre" assim que
                                                -- os agentes forem disparados.

        com_initsend()                          -- inicializa a comunicação

        msg= [[ a = com_mytid                   -- "loop" que imprime na tela os números
                while   a > 0 do                -- entre o "task id" de cada agente e 1
                        print(a)
                        a = a - 1
                end
             ]]
        com_exc( msg)

        com_mcast( escravos, "WHILE")           -- envia um comando de "loop", com
                                                -- cabeçalho "WHILE" para todos os agentes
                                                -- contidos na tabela "escravos"
end

start()
```

Figura 2.20: Enviando um trecho de código Lua

nadas no receptor e então executadas como uma única mensagem.

Foram implementadas ainda novas funções de envio e recepção de forma a garantir que as mensagens recebidas sejam tratadas na ordem correta. Para cada par transmissor/ receptor, todas as mensagens são numeradas e nenhuma mensagem é executada no receptor se houver uma mensagem de número menor que ainda não tenha sido recebida, exceto no caso da função *com_ping*, que envia mensagens não numeradas.

No modelo dos Agentes Lua não é feito nenhum tipo de controle para corrigir perda de mensagens, e o protocolo utilizado internamente – UDP – também não corrige estas perdas. Os agentes apenas conseguem sinalizar a perda de uma mensagem, mas não recuperá-la.

Além da criação destas funções, surgiu a necessidade de haver um agente mestre que gerencia os demais agentes. Esta necessidade deve-se a uma alteração feita na biblioteca dos agentes-lua (mecanismo básico). Achamos que seria interessante não fixar uma máquina e uma porta para cada agente, mas sim fixar apenas a máquina e, caso a porta especificada estiver sendo utilizada, procurar uma porta disponível para que o agente possa ser disparado.

Uma vez que a configuração dos agentes passou a poder ser alterada dinamicamente, tornou-se necessária a criação deste agente mestre de forma que ele concentre todas as informações dos agentes para fornecê-las aos demais agentes.

Em sua implementação atual, o agente mestre é responsável por disparar cada um dos agentes e guardar algumas informações a seu respeito. A intenção é incluir no agente mestre outras funções de gerência dos agentes, tais como sincronismo entre dois ou mais agentes e monitoramento de variáveis.

Algumas das funções disponíveis para o usuário são:

- *com_spawn*

  Dispara n agentes a serem usados na aplicação. Após disparar todos os agentes,

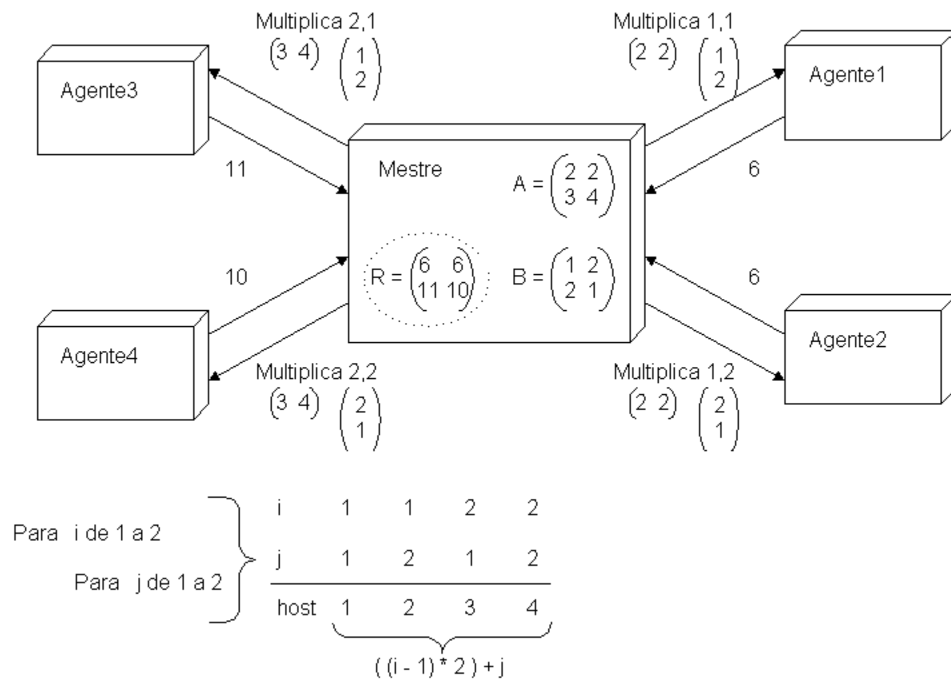


Figura 2.21: Exemplo: Multiplicação de Matrizes

chama a função *com_spawn_completed*, que deve ser definida pelo usuário. A implementação pré-definida de *com_spawn_completed* é exibir a mensagem *"Processes Ready"*.

- *com_ping*

  Verifica se um agente (*host*) está recebendo mensagens, através do envio de uma mensagem com evento de resposta. O evento de resposta vem com um comando que exibe a mensagem "*host* is alive.".

- *com_initsend*

  Inicializa estruturas de comunicação.

- *com_exc*

  Coloca um comando no *buffer* de envio.

- *com_pk*

  Coloca uma variável a ser enviada no *buffer*.

- *com_mcast*

  Envia os comandos previamente armazenados no buffer para os agentes (*hosts*) desejados. Utiliza as funções *com_sending_msg*, que é chamada antes do envio de cada

```
function start()
        com_spawn(4)
end

function com_spawn_completed( hosts)
        local i = 1

        A = { {2, 2}, {3, 4} }
        BT= { {1, 2}, {2, 1} }

        com_initsend()

        com_exc(Multdef)
        com_mcast(hosts, "MULTIDEF")

        while i <= 2 do
            local j = 1

            while j <= 2 do
                    com_pk( "linha",  A[i])
                    com_pk( "coluna", BT[j])
                    com_exc("multiplica("..i..", "..j..")")
                    com_send(hosts[((i-1)*2)+j], "MULTI
"..i.."X"..j)
                    j = j + 1
            end
            i = i + 1
        end
end

Multdef = [[ function multiplica( i, j)
                    local k, resp = 1, 0

                    while linha[k] ~=nil do
                            resp = resp + (linha[k]*coluna[k])
                            k = k + 1
                    end
                     com_initsend()
                    com_exc("Resp("..i..", "..j..", "..resp..")")
                    com_send("master", "RESP "..i.."X"..j)
            end
        ]]
```

```
h = 0
R = { {}, {} }

function Resp( i, j, x)
        h = h + 1
        R[i][j] = x
        if h == 4 then
           mostra_resultado()
           com_exit_all()
        end
end

function mostra_resultado()
        print("Matriz A")
        mostra_matriz(A)

        print("Matriz BT")
        mostra_matriz(BT)

        print("Matriz Resposta")
        mostra_matriz(R)
end



function mostra_matriz(M)
        local i, j = 1, 1

        while M[i] ~= nil do
            j = 1
            while M[i][j] ~= nil do
                  print("(" .. i .. "," .. j .. ") = " .. M[i][j])
                  j = j + 1
            end
            i = i + 1
        end
end

start()
```

Figura 2.22: Código do exemplo de Multiplicação de Matrizes

uma das mensagens, e recebe como parâmetro o agente receptor e o cabeçalho da mensagem enviada, e *com_header*, que recebe seu segundo parâmetro (o cabeçalho da mensagem). Ambas as funções podem ser redefinidas pelo usuário.

- *com_send*

  Envia os comandos previamente armazenados no buffer para o agente desejado. Utiliza as funções *com_sending_msg*, que é chamada antes do envio de cada uma das mensagens, e recebe como parâmetro o agente receptor e o cabeçalho da mensagem enviada, e *com_header*, que recebe seu segundo parâmetro (o cabeçalho da mensagem). Ambas as funções podem ser redefinidas pelo usuário.

- *com_header*

  Coloca um comando para imprimir o cabeçalho recebido como parâmetro no início do *buffer* de envio.

- *com_sending_msg*

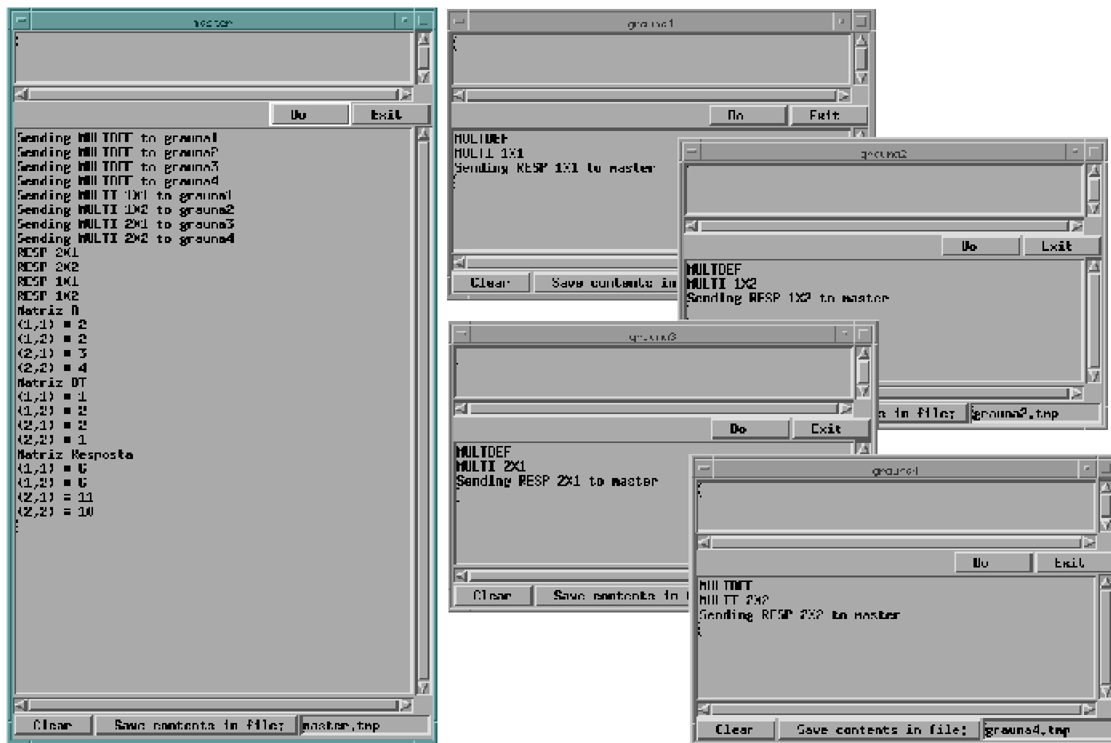


Figura 2.23: Resultado da Multiplicação de Matrizes

Mostra a mensagem "Sending "..*msg*.." to ".. *to* na console do agente que chamou a função, onde *msg* é o cabeçalho da mensagem e *to* é o agente receptor da mensagem.

- *com_exit*

  Finaliza a execução de todos os agentes contidos na tabela que é passada como parâmetro para esta função.

2.4.2 Extensão para C

Para introduzir um ganho maior no desempenho deste modelo, foram feitas algumas funções para armazenar em um buffer dados que devem ser tratados exclusivamente pela parte da aplicação escrita na linguagem C.

A extensão cria um mecanismo auxiliar que evita que os dados transmitidos pela parte da aplicação escrita em C (e que serão tratados por funções também escritas em C), tenham que ser transformados em valores Lua, para somente então serem lidos pelas funções C.

O funcionamento destas funções é bem similar ao descrito anteriormente para funções Lua. As novas funções apenas colocam mensagens em um novo *buffer*, exclusivo para mensagens que não sejam código Lua. Além destas funções, existem as funções associadas a elas que, uma vez que o *buffer* é recebido por um agente, retiram o conteúdo do *buffer* recebido. As funções para o envio do *buffer*, assim como as demais funções do mecanismo,

permanecem as mesmas. Desta forma o usuário pode montar comandos Lua em um *buffer* e pode colocar dados C no outro *buffer*. Porém, ao enviar uma mensagem a um agente, ambos os *buffers* serão enviados. Os comandos do *buffer* Lua serão executados e, entre eles deve haver a chamada de uma função C, que irá ler os dados recebidos pelo *buffer* C.

Quando colocamos elementos no *buffer* de envio, através de código C, passa a ser relevante saber qual o tipo do dado (elemento) que está sendo transmitido, ao contrário do que acontece em Lua. Para isso, foi necessário disponibilizar funções que colocam elementos específicos no *buffer* como um inteiro, um *float* ou ainda uma *string*. Além disso, nesta nova visão do mecanismo, o usuário deve ainda dizer qual a função C que irá tratar estes dados exclusivamente C.

Exemplo

```
...
  /*
  ** TESTE -- Envia um inteiro, um float e uma string para o agenteB,
  **          que deve conter em seu código a função foo registrada.
  */
  com_initsend();
     com_pkint(3);
     com_pkfloat(5.65);
     com_pkstr("Teste");
  com_cput("foo()");           /* comando Lua que será executado */
                               /* no agente receptor.            */
  com_send("AgenteB", "TESTE");
...
```

Figura 2.24: Código C do Agente A.

No exemplo da figura 2.24, o agente *a* coloca no *buffer C* o inteiro 3, o float 5.65 e a string "Teste", indica que a função do agente *b* – receptor – que irá tratar a mensagem é a função `foo`[7] e envia a mensagem com o cabeçalho "TESTE".

A função *com_cput* recebe como parâmetro um comando Lua que será executado no agente receptor. Normalmente este comando é a chamada de uma função C a qual irá se encarregar de ler todos os dados transmitidos por um outro agente.

Note que o *buffer* enviado para o agente *b* conterá tudo o que foi armazenado desde o último *send* — *com_send* ou *com_mcast* — ou desde o último *com_initsend*, incluindo qualquer comando que tenha sido armazenado no *buffer* através de um comando de Lua.

Um *buffer* C é construído em paralelo com o *buffer* de comandos Lua, contendo apenas as informações armazenadas através dos comandos disponíveis na extensão C, tais como *com_pkint*, *com_pkfloat* e *com_pkstr*. Quando a função *com_cput* é chamada, é executada uma função similar à *com_exc* que coloca o comando passado como parâmetro no *buffer*

[7] A função `foo` deve ter sido registrada em Lua.

Lua e além disso, sinaliza, como será visto mais adiante, que existe um conteúdo C que também deverá ser transmitido.

A função *com_send* (ou *com_mcast*) pode ser chamada tanto de Lua quanto de C, e sua execução irá transmitir o conteúdo de ambos os *buffers*.

No agente *b*, a função *foo* deve estar definida e registrada em Lua — o parâmetro passado na função *com_cput* é um comando Lua qualquer — como mostra a figura 2.25.

```
...
lua_register("foo", teste)
...
/*
** TESTE -- Lê inteiro, um float e uma string enviados pelo agenteA.
*/
void teste()
{
int   i;
float f;
char  s[10];

   com_unpkint(&i);
   com_unpkfloat(&f);
   com_unpkstr(s);
}
```

Figura 2.25: Código C do Agente B.

Observe que os dados a serem lidos devem ser desempacotados (retirados do *buffer* pelo agente receptor) na mesma ordem em que foram empacotados (colocados no *buffer* pelo agente transmissor), através das funções *com_unpkint*, *com_unpkfloat* e *com_unpkstr*.

Para implementar esta extensão do mecanismo, duas formas distintas de lidar com estes dados C foram estudadas.

- Transmissão dos dados, através de comandos Lua:

  A primeira forma de enviar o buffer C armazenava todos os dados a serem enviados (numa representação de *strings*) num *buffer* auxiliar.

  Quando o agente que queria enviar a mensagem chamava a função de envio, todo o *buffer* de envio era considerado uma única *string* que era colocada no *buffer* Lua como parâmetro da função *com_crecv*. O *buffer* Lua era então enviado para o agente receptor (figura 2.26).

```
com_exc('com_crecv("3 5.65 Teste")')

com_exc('foo()')
```

Figura 2.26: Transmissão dos dados como strings.

No agente receptor, a função *com_crecv* armazenava a *string* recebida em um *buffer* de entrada, que mais tarde seria consultado por uma função C. Esta função C deveria então fazer uso das funções *com_unpkint*, *com_unpkfloat* e *com_unpkstr*, para desempacotar os dados do *buffer* de entrada.

- Transmissão do buffer C em separado:

  Outra forma de enviar o buffer C não transformava os dados a serem enviados em comandos Lua, para que o interpretador Lua não precisasse lê-los. Ao invés disso, o buffer C é enviado separadamente. Todas as mensagens enviadas são separadas em duas partes, uma Lua e outra C. A parte C é colocada diretamente em um *buffer* de entrada C e a outra parte é então interpretada.

  Uma mensagem pode não conter sua parte C, no caso de nenhum dado ter sido colocado no *buffer*, porém o inverso não ocorre. Mesmo que o usuário envie vários blocos consecutivos de mensagens apenas com dados C, existirá sempre uma parte desta mensagem que será um código Lua. Isto deve-se ao fato do mecanismo ter um controle de numeração de mensagens, que insere código Lua nas mensagens transmitidas para verificar se nenhuma mensagem foi perdida. Além disso, a chamada da função que irá tratar este buffer C será feita através de um comando Lua, que foi armazenado no *buffer* através do comando *com_cput*. Sem este comando, os dados C transmitidos não poderiam ser lidos no receptor, apesar deste conter todos os dados em seu *buffer*.

Note que apesar de haver maneiras distintas de enviar os dados C, a maneira como o usuário envia e recebe estes dados permanece a mesma, como pode ser observado nas figuras 2.24 e 2.25. Ou seja, a forma como estes dados C estão sendo transmitidos fica transparente para o usuário.

Alguns testes foram feitos no sentido de avaliar qual das formas de enviar os dados C seria mais eficiente. A forma escolhida para fazer parte do mecanismo de comunicação estendido dos agentes Lua foi a segunda, pois uma vez que o interpretador Lua não precisa sequer avaliar os dados transmitidos, esta forma torna-se mais eficiente.

Capítulo 3

Problemas Estudados

Para avaliar a eficiência do mecanismo dos Agentes Lua, proposto no capítulo anterior, fizemos uma análise de desempenho deste mecanismo, utilizando para isto duas aplicações. O objetivo desta análise é saber como se comporta este mecanismo orientado a eventos em relação a um modelo cliente/servidor tradicional, já que estes modelos são significativamente diferentes. Para esta análise foram escolhidas duas aplicações, já implementadas em PVM [Sun90], com características bastante diferentes.

A primeira aplicação é uma simulação do problema dos N-Corpos [FG93]. Esta aplicação lida com uma estrutura de dados muito grande, que é transmitida entre todos os agentes diversas vezes durante a execução da aplicação.

Já a segunda, que é sobre o problema de Visualização Volumétrica Distribuída [dBS97], também lida com uma estrutura de dados bastante grande, porém a comunicação fica quase que restrita a pares de agentes.

As aplicações escolhidas foram implementadas com os Agentes Lua. Cabe destacar que não houve necessidade de reimplementar toda a aplicação, mas apenas a parte de comunicação.

A proposta de escrever a parte estrutural da aplicação em Lua também é uma opção interessante, pois torna seu código mais maleável, facilitando alterações na parte estrutural do problema.

As aplicações foram escolhidas por apresentarem diferentes características, permitindo uma melhor avaliação do comportamento do mecanismo dos Agentes Lua. Apesar de ambas as aplicações lidarem com uma estrutura de dados muito grande, no problema dos N-Corpos o tamanho das mensagens trocadas entre os processos é, tipicamente, maior. Além disso, no problema dos N-Corpos, devido ao fato de todos os processos estarem se comunicando entre si, aparece um problema de sincronismo entre os diversos processos. Outra característica distinta entre as aplicações é que no problema da Visualização Volumétrica Distribuída, ao aumentarmos o número de processadores, o tamanho das mensagens diminui, enquanto no problema dos N-Corpos pode-se dizer que o tamanho das mensagens permanece constante.

Os algoritmos de cada uma das aplicações tiveram que sofrer algumas alterações, uma vez que o modelo dos Agentes Lua (orientado a eventos) é significativamente diferente do modelo cliente/servidor usado em PVM.

3.1 Problema dos N-Corpos

O problema dos N-Corpos, utilizado como um dos exemplos do mecanismo proposto, foi baseado no trabalho [FG93], que implementou este problema em PVM [Sun90].

O objetivo deste problema é estudar a evolução de um sistema de corpos sob a influência da atração gravitacional Newtoniana. Os corpos consistem de uma massa, posição e velocidade inicial e estão distribuídos sobre um domínio físico finito. A simulação segue sobre um número de iterações, onde, em cada iteração, há o cálculo das forças em cada corpo e há a atualização de sua posição e de outros atributos. A implementação da simulação dos N-Corpos é baseada no algoritmo de *Barnes-Hut* [CT92, BH86, Vel94], porém com uma árvore binária.

Este problema é adequado para cálculos astrofísicos. Assume-se um universo 2-D, que é um retângulo de dimensões XMAX x YMAX, limitado pelos eixos. As partículas neste universo são definidas por uma massa, posição e velocidade.

O cálculo resultante das forças tem complexidade média $O(n^2)$ já que cada partícula é afetada pelo movimento das outras partículas. O algoritmo de Barnes-Hut reduz a complexidade média para $O(n \, log \, n)$ usando o seguinte princípio: um grupo de corpos que está longe de uma partícula i pode ser modelado usando uma única partícula j cuja massa é a massa total do grupo de corpos e cuja posição é o centro de massa do grupo.

Em cada passo da simulação, uma árvore quaternária é construída usando as partículas do passo anterior. Inicialmente, a árvore está vazia (isto é, não contém partículas). Como as partículas são adicionadas à árvore, o espaço do problema é recursivamente dividido em quatro partes e essa divisão é feita até que não mais de uma partícula esteja presente em cada divisão do espaço. Como as partículas são adicionadas na árvore, nós intermediários são usados para acumular a massa total e o centro de massa de um grupo de partículas.

Depois que a árvore é construída, a aceleração de uma partícula é calculada descendo na árvore e aplicando o critério de precisão em cada nó. Se um nó provê uma aproximação suficientemente exata dos efeitos em uma partícula, então a aceleração é calculada pela interação do nó e da partícula. Caso contrário, a contribuição do nó é calculada pela somatória das contribuições dos quatro filhos do nó.

Como foi mencionado anteriormente, o algoritmo de Barnes-Hut usa uma abordagem hierárquica e resulta em uma complexidade média $O(n \, log \, n)$. Para conseguir esse valor de complexidade, toda a informação a respeito das partículas é representada na forma de árvore quaternária. Pode-se, contudo, usar uma árvore binária, no lugar da árvore quaternária, para representar as partículas, sem aumentar a complexidade computacional do algoritmo. Com essa mudança, o procedimento *Orthogonal Recursive Bisection* (ORB), responsável pela formação de uma árvore binária de corpos, pode ser usado. A implementação paralela discutida nesse trabalho utiliza árvores binárias, que são formadas usando o ORB.

Formulações paralelas do algoritmo de Barnes-Hut envolvem o particionamento do domínio (ou da árvore) entre vários processadores com o objetivo de otimizar o problema seqüencial. As partículas são divididas entre os processadores de modo que eles tenham uma carga computacional inicialmente equivalente.

A construção da árvore é simplificada por duas características: todas as partículas em um sub-domínio particular são designadas a um único processador; e, selecionado o número de partições em qualquer dimensão do domínio para ser uma potência de 2, cada sub-domínio corresponde na verdade a um nó na árvore de Barnes-Hut. Na estrutura da árvore resultante, as folhas representam as partículas e os nós intermediários representam grupos de partículas.

A construção da árvore é feita nos dois passos seguintes: (a) construção de uma árvore local; (b) fazer a união de árvores locais para formar uma árvore global. A construção da árvore local não requer nenhuma comunicação e pode ser feita por todos os processadores independentemente. Para cada sub-domínio, associado a um processador, é construída uma árvore correspondente. As dimensões do nó raiz na árvore são as do sub-domínio e não do domínio inteiro.

O passo seguinte associa com cada nó, o centro de massa e a massa equivalente das partículas contidas na sub-região do nó. Isso é feito pela propagação da informação das folhas para os nós pais, e assim sucessivamente, até a raiz da árvore. Depois desse passo, as folhas passam a conter as informações a respeito das partículas e os nós internos passam a conter informações sobre os grupos de partículas.

Depois da propagação das informações, a árvore pode ser usada para calcular a força resultante em cada partícula. O algoritmo que calcula a força que age em cada partícula começa na raiz da árvore e segue até chegar nas folhas. Em cada nó interno, o cálculo é feito para determinar se o agregado de partículas, representado pelo nó, pode ser considerado como único nó. Isso é feito calculando a razão entre o tamanho do sub-domínio que está sendo processado, *tam (nó)*, e a distância entre a partícula e o centro de massa do sub-domínio, *dist(nó, corpo)*. Se o sub-domínio está muito longe da partícula (isto é, se *tam (nó) / dist(nó, corpo)* $\leq \theta$, onde θ é um parâmetro fixo de precisão), a força Newtoniana entre a partícula e o sub-domínio é calculada. Caso contrário, o algoritmo é aplicado recursivamente para cada filho do nó atual.

Depois do cálculo das forças, as posições e as velocidades das partículas são atualizadas. Em seguida, é feita a união das árvores locais, onde apenas os nós raízes das sub-árvores locais precisam participar.

Dado que os corpos têm novas posições, uma nova árvore deve ser formada e todo o procedimento é repetido para a nova iteração. A simulação segue por um número pré-determinado de iterações.

3.1.1 Descrição da Aplicação

A aplicação consiste em um processo mestre e um conjunto de processos simuladores. O processo mestre lê os dados de inicialização e cria os processos simuladores (figura 3.1) os quais executam o núcleo da simulação (figura 3.2).

O universo é dividido em domínios e cada processo simulador é alocado a um domínio e a todas as partículas deste domínio. Cada simulador fica responsável por toda computação associada com suas partículas. A simulação prossegue com algumas iterações, nas quais a força resultante em cada partícula é computada e a posição da partícula é atualizada.

1. Faz as declarações iniciais, seta parâmetros pré-definidos;
2. Se um arquivo chamado "input" existir, lê os parâmetros iniciais – número de partículas para a simulação, número de nós, etc;
3. Cria os processos simuladores;
4. Envia os parâmetros iniciais para os simuladores;
5. Lê a distribuição inicial de partículas;
6. Envia partículas para o simulador 0;
7. Espera que os simuladores terminem e coleta as posições finais das partículas.

Figura 3.1: Pseudocódigo do mestre

São utilizadas duas estruturas de dados principais:

1. uma lista com as partículas;
2. uma árvore binária com as partículas distribuídas de forma balanceada.

Cada partícula tem um identificador, uma posição (coordenadas x e y), massa e velocidade (componentes x e y), além de outros campos diversos.

A distribuição das partículas é representada em forma de árvore, que é formada utilizando a técnica da Bisseção Ortogonal Recursiva (ORB). A raiz da árvore representa todo o domínio, o qual é bissecionado recursivamente em sub-domínios, representados em forma de árvore. As folhas da árvore representam partículas. Os nós intermediários representam sub-domínios e as partículas nos sub-domínios. Cada nó da árvore é representado por uma estrutura de dados que contém o nível da árvore, direção do bissetor, posição do centro de massa (coordenadas x e y), massa total e outros campos.

A técnica Bisseção Ortogonal Recursiva (ORB) apresenta as seguintes características :

- A raiz aponta para a raiz da árvore, para todos os processos simuladores. A raiz representa o universo e todas as partículas simuladas (figura 3.5).
- O universo é dividido em domínios (sub-árvores)e cada processo simulador é responsável por um domínio (figura 3.6).

O trabalho [RR98] apresentou um estudo sobre a simulação dos N-Corpos e concluiu que os experimentos feitos com menos de 4.096 partículas não mostravam uma paralelização vantajosa em relação à simulação seqüencial. Nestes casos, o tempo gasto com processamento é muito pequeno, o que aumenta o custo relativo da fase de comunicação.

1. Recebe parâmetros do mestre;

2. Se simulador = 0, recebe distribuição inicial das partículas;

3. Distribui as partículas utilizando o algoritmo ORB – todos os simuladores participam (fig. 3.3)[1];

4. Para iteração = 1 até iteração = MAX_ITER

   (a) Forma árvore com as partículas locais (fig. 3.4);
   (b) Troca informações das árvores de partículas com outros simuladores (fig. 3.5);
   (c) Computa os limites dos domínios dos processos;
   (d) Computa a força em cada partícula local;
   (e) Decompõe a árvore e coleta as partículas. Se a partícula atravessou os limites do processo, a envia para o processo apropriado.

5. Envia distribuição final de partículas para o mestre e termina.

Figura 3.2: Pseudocódigo dos Processos Simuladores

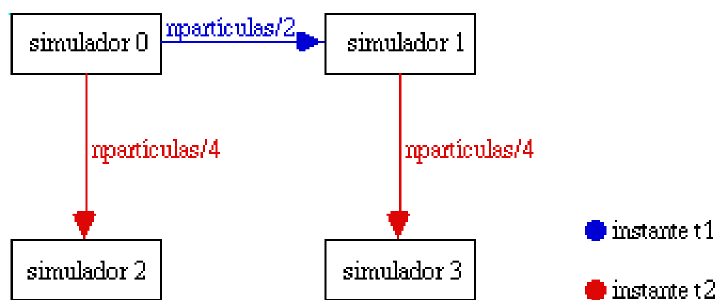


Figura 3.3: Distribuição das Partículas (ORB)

3.1.2 Implementação com os *Agentes Lua*

Para implementar a aplicação dos n-corpos em Lua, foram necessárias algumas alterações no algoritmo original, uma vez que este se baseava num modelo *send/receive* e o modelo que se desejava utilizar era um modelo orientado a eventos, como apresentado no capítulo 2.

A maioria das alterações necessárias foram bem simples e limitaram-se a trocar algumas iterações por um modelo semelhante a uma máquina de estados, como mostrado na figura 2.6.

Para os itens `4.b` e `4.e` da figura 3.2, foi necessária a implementação de um sincronismo entre os agentes simuladores, pois a execução de cada um desses passos requer que todos os agentes tenham completado o passo anterior.

No caso do item `4.b`, em que os agentes coletam todas as árvores de partículas de cada um dos outros agentes, formando uma única árvore de partículas, cada passo da troca de mensagens é numerado e, desta forma, cada agente sabe com quem ele deve trocar informações.

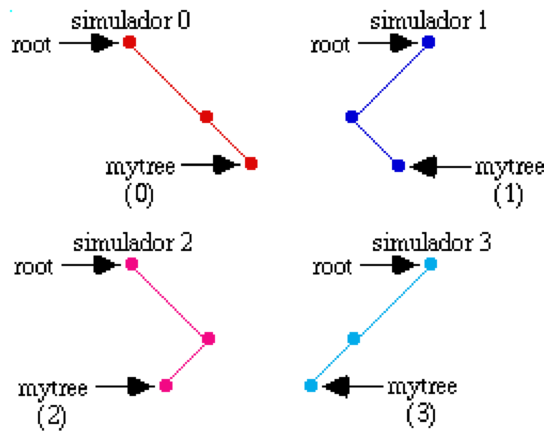


Figura 3.4: Árvores de Partículas Locais

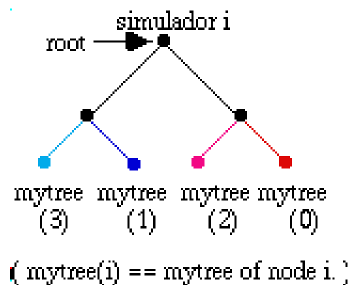


Figura 3.5: Árvore de Partículas

A tabela 3.1 indica, para cada passo da troca de um processo, com que processo simulador a árvore deve ser trocada. Para n=4, onde `n` é o número de agentes, serão necessários apenas dois passos para que todos os simuladores contenham a árvore completa. No primeiro passo, cada simulador obtém a informação de seu processo vizinho, e fica portanto com a informação correspondente a duas árvores. No passo seguinte, cada troca corresponde à informação de duas árvores (a que o simulador já tinha e a obtida no passo anterior). Ao final de $log(n) - 1$ passos, cada um dos simuladores contém as informações de todos os outros simuladores (n árvores)[2].

Desta forma, quando chega uma mensagem de um agente simulador que não está no mesmo passo que o agente receptor, ele envia uma mensagem de volta ao agente que enviou a mensagem, com uma chamada para a função que irá enviar novamente a mensagem, até que ela possa ser respondida. Esta solução poderia ser melhorada sincronizando todos os agentes ao final de cada passo da troca das árvores.

Em PVM este sincronismo não é necessário, pois uma vez que a cada passo os processos sabem com quem eles querem trocar informações, é possível, com primitiva de *receive*, escolher qual a mensagem que se deseja receber. No caso de chegar uma mensagem indesejada, ela fica num *buffer* até o momento de ser consumida.

[2]Ver figuras 3.4 e 3.5.

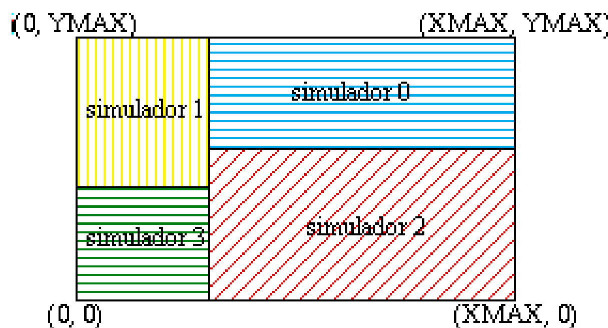


Figura 3.6: Domínio dos Nós

Nó / Passo	0	1	...	log(n)-1
0	1	2	...	2ˆpasso
1	0	3	...	
2	3	0	...	
3	2	1	...	
...	...	...	...	
n-1	n-2	n-3	...	

Tabela 3.1: Troca das árvores de partículas entre os diversos agentes.

Já no caso do item `4.e`, em que cada agente decompõe a árvore e coleta as partículas, enviando as partículas que atravessaram os limites para os agentes apropriados, cada agente realiza os seguintes passos para sincronizar todos os agentes (ver figura 3.7) :

- inicializa um contador, no início do programa principal, de modo que todos os agentes possuam seus contadores zerados.

- envia uma mensagem, para cada um dos agentes, incluindo o próprio, que incrementa o contador e verifica se este é igual ao número de agentes a serem sincronizados. Caso o contador seja igual ao número de agentes, o contador é zerado novamente e pode considerar todos os agentes sincronizados.

Este sincronismo foi utilizado antes dos agentes começarem a enviar as partículas que atravessaram seus limites, pois uma vez que as mensagens são assíncronas, poderia chegar uma mensagem com uma nova partícula antes que um agente terminasse de trocar todas as informações de suas árvores com os demais agentes simuladores (item 4.b).

Uma vez que cada agente não sabe quantas partículas irá receber, é necessário fazer este sincronismo novamente quando cada um dos agentes terminar de enviar suas partículas, para que os agentes possam começar o passo seguinte.

A idéia é fazer, no futuro, uma função de sincronismo embutida na própria camada de comunicação dos Agentes Lua.

Além do problema de sincronismo, outro ponto que merece destaque é a limitação da capacidade do *buffer* de recepção. Como foi dito no capítulo anterior, a capacidade do *buffer*

```
--   runaway_sync = 0;  No início do programa principal.

function sync_to_runaway()
  local msg

  msg = [[
         runaway_sync = runaway_sync + 1
         if runaway_sync == nnodes then
            runaway_sync = 0
            node_runaway()                -- pronto para o item 4.e
         end
        ]]
  com_exc(msg)
  com_mcast( node_tids, "RUNAWAY_SYNC " .. mynum)
end
```

Figura 3.7: Sincronismo entre os agentes.

dos Agentes Lua não é ilimitada e, além disso, não é implementado nenhum mecanismo de controle de fluxo. Os agentes conseguem apenas detectar a perda de uma mensagem, porém não conseguem recuperá-la (a menos que seja implementado um controle na própria aplicação).

Na versão dos Agentes Lua, na rotina de troca de informações das árvores de partículas do item `4.b` (figura 3.2), dois agentes no mesmo passo não podem enviar mensagens ao mesmo tempo, pois neste caso, quando a aplicação está executando com um número grande de partículas (mais de 1024 partículas) as mensagens enviadas estouram o *buffer* de recepção[3], e, como o agente receptor também está ocupado enviando mensagens, ele não consegue retirá-las do *buffer* a tempo. Isto ocorre devido ao problema descrito na seção 2.3.

No apêndice deste trabalho, as seções A.1.1 e A.1.2 mostram trechos de códigos que exemplificam o uso dos agentes para o problema dos N-Corpos.

3.2 Visualização Volumétrica Distribuída

O problema de visualização volumétrica, utilizado como o segundo exemplo do mecanismo proposto, foi baseado no trabalho [dBS97], que implementou este problema em PVM [Sun90].

A implementação deste problema usa uma otimização da técnica de *Ray-Casting*. *Ray-Casting* é uma técnica muito usada em visualização volumétrica para visualização de imagens médicas. A imagem é dividida entre os processadores e cada um deles fica responsável por uma partição do espaço da imagem.

[3]Em algumas plataformas, como o Linux, não é possível aumentar o tamanho máximo do *buffer* de recepção.

3.2.1 Descrição do Problema

O problema consiste em visualizar um conjunto de dados tri-dimensionais. Como já foi dito, a técnica utilizada foi a técnica de *Ray-Casting*.

Nesta implementação, o volume de dados (uma imagem) é dividido entre os processadores. Cada processador obtém uma imagem intermediária com a partição do volume de sua responsabilidade e executa as mesmas tarefas em diferentes regiões da imagem. O processador responsavél pela exibição precisa combinar as sub-imagens para obter a imagem final (figura 3.8).

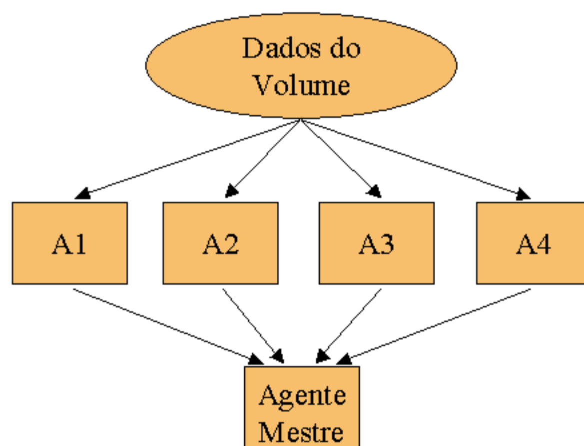


Figura 3.8: Comunicação entre os Agentes.

Neste particionamento, os processadores dividem entre si os *pixels* da imagem para gerar a imagem completa. Com este tipo de particionamento é mais simples se obter um balanceamento uniforme, pois a duplicação do volume de dados pelos processadores permite que se elimine quase totalmente a comunicação entre os processadores. Esta comunicação fica restrita ao processador responsável pela exibição da imagem e cada processador. Este tipo de particionamento pode ser feito de forma estática ou dinâmica e de várias maneiras: *pixels* individuais, blocos de *scanlines* ou blocos retangulares.

Inicialmente, o agente mestre (visualizador) envia alguns parâmetros para os demais agentes. Em seguida, um destes agentes fica responsável apenas pelo particionamento da imagem e sua distribuição entre os demais agentes que irão calcular o volume e enviar os resultados diretamente para o agente mestre. Deste modo, o processo visualizador pode ficar dedicado a exibir a imagem volumétrica (figura 3.9).

No particionamento estático, a imagem é dividida em blocos de *scanlines* ou blocos retangulares contíguos, que são atribuídos a cada processador. Na figura 3.10a, o número de *scanlines* é igual ao número de processadores (4 no exemplo). Quando o particionamento em blocos retangulares é feito, o número de blocos é igual ao quadrado do número de processadores, como pode ser observado na figura 3.10b.

Já no particionamento dinâmico, a imagem é dividida em *pixels* de *scanlines* (figura 3.11) ou blocos retangulares contíguos (figura 3.12). É feito então um particionamento

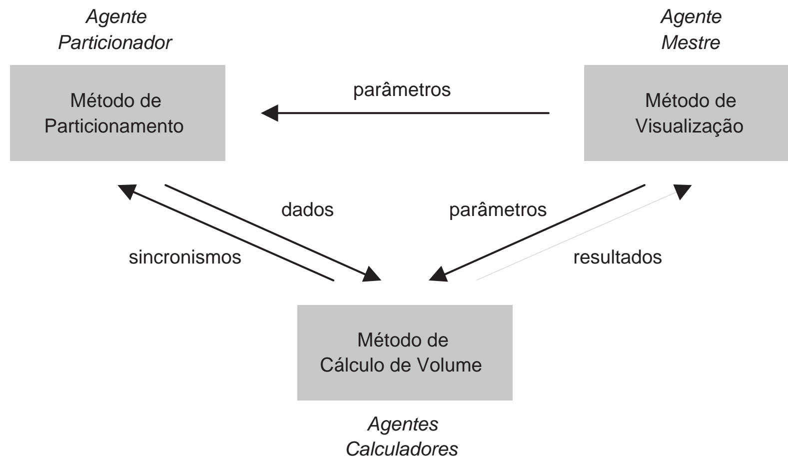
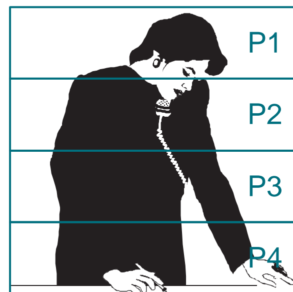
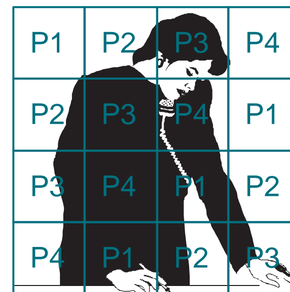


Figura 3.9: Estrutura da Aplicação.



(3.10a) blocos de *scanlines*



(3.10b) blocos retangulares

Figura 3.10: Particionamento estático.

estático, até o número de processadores, e, a partir de então, inicia-se o particionamento dinâmico, onde a medida que os processadores terminam de fazer o cálculo volumétrico, o processador responsável pelo particionamento lhes envia novas sub-imagens a serem calculadas.

3.2.2 Implementação com os *Agentes Lua*

As alterações feitas na implementação da aplicação da Visualização Volumétrica Distribuída ficaram restritas ao problema apresentado na figura 2.6, que limitam-se a trocar algumas iterações por um modelo semelhante a uma máquina de estados. Estas alterações fizeram-se necessárias, pois o algoritmo original baseava-se num modelo *send/receive* e o modelo que se desejava utilizar era um modelo orientado a eventos, como apresentado no capítulo 2.

No caso do particionamento estático por *pixels* pôde ser observado o problema relatado na seção 2.3, onde o processo particionador envia apenas uma mensagem aos processos que calculam o volume, com toda a informação sobre quais as sub-imagens que deverão ser

Figura 3.11: Particionamento dinâmico por *scanlines*.

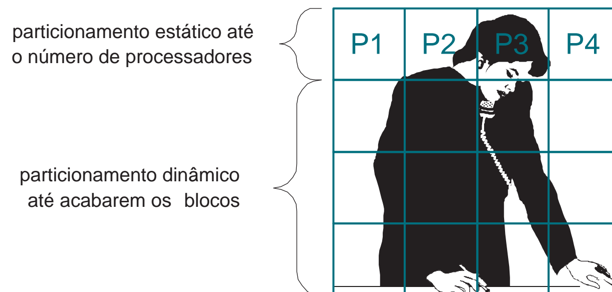


Figura 3.12: Particionamento dinâmico por blocos.

calculadas por cada processo. Desta forma, os processos que calculam o volume entram em *loop* calculando os volumes e enviando os resultados para o processo visualizador. O que ocorre, então, é que no caso da imagem ser muito grande, o processo visualizador deve ser n – onde n é o número de processos calculadores – vezes mais rápido que os demais processos para que não comece a acumular mensagens em seu *buffer* de recepção e não ocorra perda de mensagens. Cabe aqui lembrar que no modelo dos Agentes Lua não é feito nenhum tipo de protocolo para corrigir perda de mensagens, e o protocolo utilizado – UDP – também não corrige estas perdas. Os agentes apenas conseguem sinalizar a perda de uma mensagem, mas não recuperá-la.

A figura 3.13 mostra o pseudo-código dos algoritmos dos Agentes Visualizador, Particionador e Calculadores.

As figuras 3.14, 3.15 e 3.16 mostram trechos de códigos que exemplificam o uso dos agentes para o particionamento por *scanlines* dinâmico.

```
VISUALIZADOR
------------
------------

loop até receber todas as sub-imagens
   recebe sub-imagem
   mostra sub-imagem na tela
fim

(observe que o visualizador não sofre alteração de acordo com o modo de
particionamento que está sendo utilizado (estático ou dinâmico)).

PARTICIONADOR ESTÁTICO
----------------------
----------------------

Para cada um dos n processadores
   envia todas as sub-imagens em "imagem/n"
fim


CALCULADORES ESTÁTICOS
----------------------
----------------------

recebe informação de todas as sub-imagens a serem calculadas

loop ate acabarem todas as sub-imagens
   calcula sub-imagem
   envia para o visualizador
fim

PARTICIONADOR DINÂMICO
----------------------
----------------------

Para cada um dos n processadores
   envia uma sub-imagem
fim

loop ate acabarem todas as sub-imagens
   recebe sinal de processador disponível
   envia uma sub-imagem
fim

CALCULADORES DINÂMICOS
----------------------
----------------------

loop
   recebe uma sub-imagem
   calcula sub-imagem
   envia para o visualizador
   envia sinal de processador disponível para particionador
fim
```

Figura 3.13: Algoritmos dos Agentes Visualizador, Particionador e Calculadores.

```
...
        case ScanlineDinamico:
            /*
            ** DVV_RESULT_SCANLINE
            */
            com_unpkint(&scanline);
            com_nunpkbyte(&tela[(Tela_Y-1-scanline)*Tela_X], Tela_X);

            cdPutImageRGB(Tela_X, 1,
                          &tela[(Tela_Y-1-scanline)*Tela_X],
                          &tela[(Tela_Y-1-scanline)*Tela_X],
                          &tela[(Tela_Y-1-scanline)*Tela_X],
                          0, Tela_Y-1-scanline, Tela_X, 1);
            cdFlush();

            dvv_result_i++;
            break;
...
```

Figura 3.14: Trecho de Código do Agente Visualizador.

```
...
   for (scanline = 0, i = 0; i < num_proc; i++, scanline++)
   {
        char tmp[200];

        /* DVV_QUERY */
        com_initsend();
           com_pkint(scanline);
        com_cput("din_query_scanlines()");
        sprintf( tmp, "com_send(dvv_hosts[%d], [[DVV_QUERY_SCANLINE]])", i+1);
        lua_dostring(tmp);
   }  /* for num_proc */
   scanline = num_proc;
   wait_counter=0;
...

void sd_wait ()
{
   int   i;
   char  tmp[200];

   /* DVV_IDLE */
   com_unpkint(&who);

   if (scanline < num_scanlines)
   {
      /* DVV_QUERY */
      com_initsend();
         com_pkint(scanline);
      com_cput("din_query_scanlines()");
      sprintf( tmp, "com_send(dvv_hosts[%d], [[DVV_QUERY_SCANLINE]])", who+1);
      lua_dostring(tmp);
      scanline++;
   }
   else
   {
      wait_counter++;
      if (wait_counter == num_proc)
      {
         /* Encerra o loop de Queries */
         for (scanline = -1, i = 0; i < num_proc; i++)
         {
            /* DVV_QUERY */
            com_initsend();
               com_pkint(scanline);
            com_cput("din_query_scanlines()");
            sprintf(tmp, "com_send(dvv_hosts[%d],[[DVV_QUERY_SCANLINE]])", i+1);
            lua_dostring(tmp);
         }
         com_send(viewer_name, "DVV_DONE");
      }
   }
}
```

Figura 3.15: Trecho de Código do Agente Particionador.

```
void din_query_scanlines()
{
   int          me;
   int          x;
   float        xX, xY, xZ, yX, yY, yZ;

        /*
        ** DVV_QUERY                   Recebe pixel a calcular
        */
        com_unpkint(&scanline);

        if (scanline >= 0)
        {
           yX = (y_incX * scanline);
           yY = (y_incY * scanline);
           yZ = (y_incZ * scanline);
           for (x = 0; x < Tela_X; x++)
           {
              xX = (x_incX * x);
              xY = (x_incY * x);
              xZ = (x_incZ * x);
              P1[iX] = Plbn[iX] + xX + yX;
              P1[iY] = Plbn[iY] + xY + yY;
              P1[iZ] = Plbn[iZ] + xZ + yZ;
              P2[iX] = Plbf[iX] + xX + yX;
              P2[iY] = Plbf[iY] + xY + yY;
              P2[iZ] = Plbf[iZ] + xZ + yZ;

              result[x] = (char) raycast(P1,P2);
           } /* for x */

           /*
           ** DVV_RESULT               Manda resultado para o Viewer
           */
           com_initsend();
              com_pkint(scanline);
              com_npkbyte(result, Tela_X);
           com_cput("dvv_result()");
           com_send(viewer_name, "DVV_RESULT_SCANLINE");

           /*
           ** DVV_IDLE                 Avisa ao Particionador que esta IDLE
           */
           me = lua_getnumber( lua_getglobal( "com_mytid")) - 1;
           com_initsend();
              com_pkint(me);
           com_cput("sd_wait()");
           com_send(main_tid, "DVV_IDLE");

        }  /* while scanline */
        else
           free(result);
}
```

Figura 3.16: Trecho de Código dos Agentes Calculadores.

Capítulo 4

Resultados Experimentais

Este capítulo mostra os resultados observados na execução das aplicações apresentadas no capítulo 3. Para cada uma das aplicações (N-Corpos e Visualização Volumétrica Distribuída (DVV)), são mostrados os resultados das versões Lua e PVM, segundo os mesmos critérios de observação.

Para o problema dos N-Corpos, foram feitos 18 experimentos para cada uma das versões (Lua e PVM) e, cada experimento foi realizado cinco vezes. Entende-se por experimento a execução de uma das versões da aplicação, com os mesmos dados de entrada. Por exemplo, a versão Lua da aplicação dos N-Corpos, com 2 processadores, 2.048 partículas, rodando 10 iterações de cálculo de forças.

Para o problema da Visualização Volumétrica Distribuída, foram feitos 12 experimentos, cada um deles realizado cinco vezes.

Para obter os resultados experimentais, foi utilizado o *cluster* do laboratório do TecGraf (Grupo de Tecnologia em Computação Gráfica da PUC-Rio). O *cluster* é composto de oito máquinas, conectadas através de uma rede local dedicada. Estas máquinas ficam em uma rede isolada, com o objetivo de simular uma máquina paralela. O sistema utilizado no *cluster* é o Linux. As máquinas são Pentium 166 com 32M de memória, ligadas através de uma rede de 100 Mbits/s.

4.1 Resultados do Problema dos N-Corpos

Para avaliar o problema dos N-Corpos, as duas implementações (Lua e PVM) foram observadas segundo as seguintes características:

- Número de Partículas
  - 512
  - 2.048
  - 4.096
- Número de Iterações de Cálculo das Forças

- 1
- 10
- 20

- Número de Nós Processadores
  - 2
  - 4

Devido à estrutura do algoritmo desta aplicação, o número de processadores utilizados deve ser uma potência de 2. Na verdade, este número deve ser uma potência de 2, mais 1, pois ainda temos o processador mestre, como mostrado na seção 3.1. Nos resultados são apresentados apenas os tempos de execução dos processadores simuladores, que foram medidos em milisegundos.

Como foi dito no capítulo 3, a paralelização deste problema torna-se vantajosa em relação à simulação seqüencial para experimentos feitos com mais de 4.096 partículas. Porém, o objetivo desta seção é comparar os resultados obtidos nas versões dos Agentes Lua e PVM. Neste caso, a comparação pode ser feita em qualquer região, basta que ambas as versões recebam os mesmos dados de entrada e possuam a mesma configuração.

Para uma iteração de cálculo das forças, foram feitos seis experimentos, para cada um dos mecanismos utilizados, variando o número de nós, 2 e 4, e o número de partículas, 512, 2.048 e 4.096. Cada um destes experimentos foi realizado cinco vezes e as médias de seus resultados são apresentadas na tabela 4.1.

	2 Nós		4 Nós	
	Lua	PVM	Lua	PVM
512	83	75	104	80
2.048	371	734	381	1195
4.096	1714	1294	908	1150

Tabela 4.1: Média da Execução de uma iteração (em ms).

A tabela 4.2 apresenta as médias dos resultados das cinco medidas tomadas para os mesmos experimentos anteriores, porém com 10 iterações de cálculo de forças. Para 4.096 partículas não foi possível realizar o experimento, pois ocorre o problema descrito no capítulo 2, sobre o mecanismo dos Agentes Lua não apresentar uma capacidade ilimitada de armazenamento de mensagens. Neste experimento, os agentes começam a perder mensagens, e, desta forma, não foi possível medir o tempo de execução.

Mais quatro experimentos foram feitos, desta vez com vinte iterações de cálculo de forças. A tabela 4.3 apresenta as médias dos resultados obtidos para os experimentos com vinte iterações de cálculo das forças, com 2 e 4 nós processadores e 512 e 2.048 partículas.

A cada iteração as partículas são redistribuídas entre os processadores. Uma vez que o domínio de cada um dos processadores é fixo, pode ocorrer um desbalanceamento do

	2 Nós		4 Nós	
	Lua	PVM	Lua	PVM
512	754	621	930	604
2.048	3491	3436	3763	3632

Tabela 4.2: Média da Execução de dez iterações (em ms).

	2 Nós		4 Nós	
	Lua	PVM	Lua	PVM
512	1781	1188	1896	1207
2.048	7268	6078	8098	6247

Tabela 4.3: Média da Execução de vinte iterações (em ms).

número de partículas entre eles. Isto faz com que o tamanho das mensagens se modifique a cada iteração, tornando relevante a observação de várias iterações.

As figuras 4.1 e 4.2 apresentam gráficos[1] de desempenho da aplicação em Lua e PVM, executadas com 2 nós processadores, e 1 e 20 iterações, respectivamente. Nestes gráficos pode ser observado o comportamento das versões da aplicação quanto a variação do número de partículas.

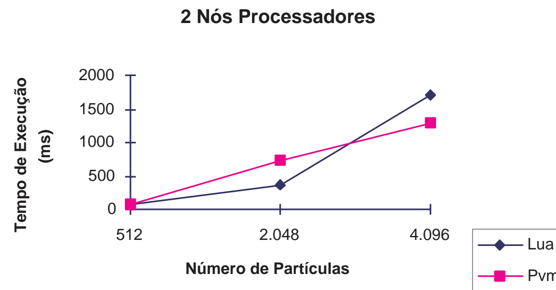


Figura 4.1: Tempo obtido para 1 iteração com 2 nós processadores.

Observando a figura 4.1, note que, surpreendentemente, a versão Lua mostrou-se mais rápida que a versão PVM para o caso de 2.048 partículas, chegando a ser 2 vezes mais rápida. Ainda nesta figura, para o caso de 512 partículas, os Agentes são apenas 11% mais lentos que PVM, enquanto para 4.096 partículas os Agentes são 32% mais lentos.

Já na figura 4.2, em ambos os casos, 512 e 2.048 partículas, os Agentes Lua são mais lentos que PVM, chegando a ser 50% mais lentos no pior caso. Neste caso, não foi possível medir o tempo da versão dos Agentes Lua para 4.096 partículas, devido a ocorrência de perda de mensagens.

As figuras 4.3 e 4.4 apresentam gráficos de desempenho da aplicação em Lua e PVM, executadas com 4 nós processadores, e 1 e 20 iterações, respectivamente. Estes gráficos mos-

[1] Todos os gráficos exibidos nesta seção são baseados nos dados apresentados na seção 4.1.

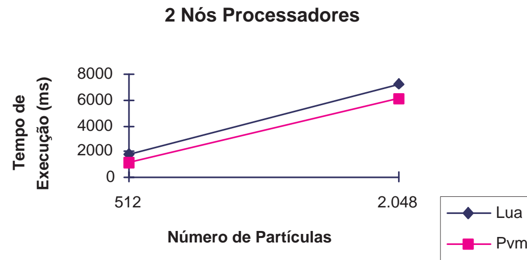


Figura 4.2: Tempo obtido para 20 iterações com 2 nós processadores.

tram o comportamento das versões da aplicação quanto a variação do número de partículas.

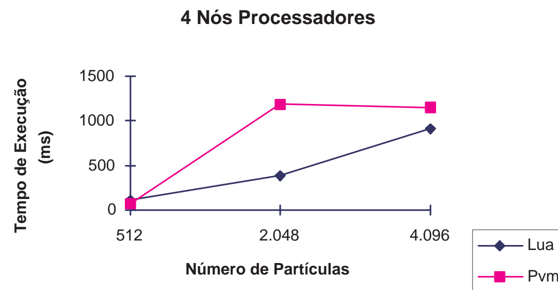


Figura 4.3: Tempo obtido para 1 iteração com 4 nós processadores.

Analisando os gráficos das figuras 4.1 e 4.3, vemos que mesmo aumentando o número de processadores utilizados, ainda podemos observar a mesma diferença de comportamento entre duas versões da aplicação. Na figura 4.3, para o caso de 512 partículas, a versão dos Agentes Lua é 30% mais lenta que a versão PVM, porém, para 2.048 e 4.096 partículas, a versão dos Agentes chega a ser 3 vezes mais rápida no melhor caso.

Na figura 4.4, tanto para 512 partículas quanto para 2.048, os Agentes perdem em desempenho, e para 512 partículas chegam a ser 57% mais lentos que a versão PVM.

Escolhemos os experimentos de 2.048 partículas para fazer uma análise mais detalhada do desempenho desta aplicação. Este caso foi escolhido por ser mais significativo, uma vez que tem um maior número de partículas que executa para todos os casos (2 e 4 processadores).

As figuras 4.5 e 4.6 apresentam gráficos de desempenho da aplicação em Lua e PVM, executadas com 2.048 partículas, e 1 e 20 iterações, respectivamente. Nestes gráficos pode ser observado o comportamento das versões da aplicação quanto a variação do número de processadores.

Neste caso, para uma iteração do cálculo das forças, a versão PVM mostra-se mais lenta que a versão dos Agentes Lua. Para o caso de vinte iterações, a versão dos Agentes Lua mostra-se mais lenta.

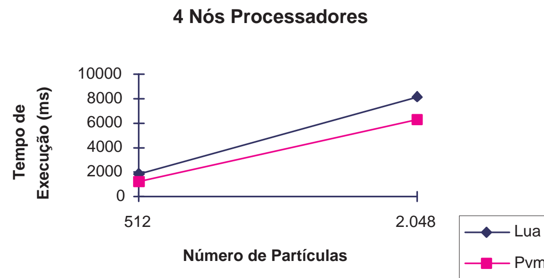


Figura 4.4: Tempo obtido para 20 iterações com 4 nós processadores.

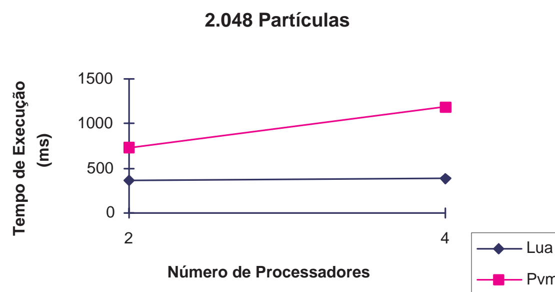


Figura 4.5: Tempo obtido para 1 iteração com 2.048 partículas.

Bons resultados, como os que podemos observar nas figuras 4.3 e 4.5, podem ser parcialmente devidos ao não tratamento de perda de mensagens. Observe que, no caso de uma iteração com quatro nós processadores, os Agentes Lua tiveram um desempenho melhor que PVM, justamente nos casos de maior número de partículas, onde há mais comunicação entre os processos.

4.2 Resultados do Problema da Visualização Volumétrica Distribuída

Para avaliar o problema da Visualização Volumétrica Distribuída, as duas implementações (Lua e PVM) foram observadas segundo as seguintes características:

- Tipo de Particionamento
  - Dinâmico por Blocos
  - Estático por Blocos
  - Dinâmico por *Scanlines*
  - Estático por *Scanlines*

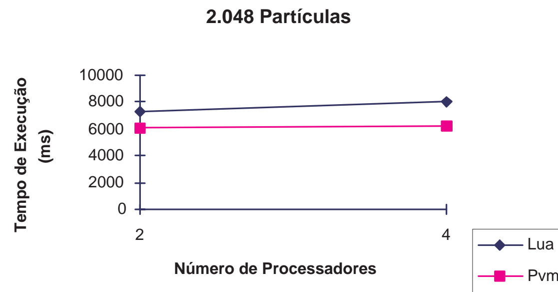


Figura 4.6: Tempo obtido para 20 iterações com 2.048 partículas.

- Número de Nós Processadores
  - 2
  - 4
  - 6

As medidas da execução desta aplicação foram tomadas em segundos e foi utilizado apenas um conjunto de dados (uma imagem), apresentado na figura 4.7. Esta imagem 3D, de dimensões 128x128x84, mostra uma cabeça humana com parte do cérebro exposto.

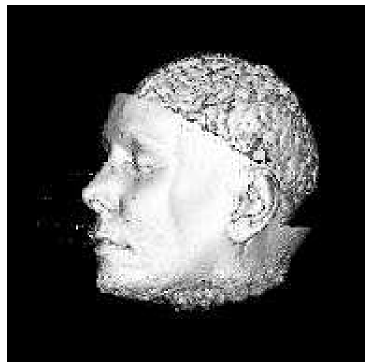


Figura 4.7: Imagem *brain*.

Para reduzir a influência do tempo de exibição, as medidas foram tomadas sem a exibição das imagens parciais, que ocorrem no refinamento sucessivo.

O tempo apresentado é o tempo total decorrido, que engloba os tempos de particionamento, cálculo, comunicação e exibição.

Para cada um dos 12 experimentos, cinco medidas foram tomadas. As médias dos resultados obtidos são apresentadas nas tabelas 4.4 e 4.5, que mostram os particionamentos por blocos e por *scanlines* respectivamente.

Ao medir os tempos do particionamento estático por *scanlines* foi observado novamente o problema de perda de mensagens. Este caso pôde ser contornado pois o problema ocorria

	Bloco Dinâmico		Bloco Estático	
	Lua	PVM	Lua	PVM
2 nós	0.362	0.285	0.347	0.280
4 nós	0.199	0.172	0.259	0.171
6 nós	0.229	0.143	0.334	0.140

Tabela 4.4: Tempos obtidos para o particionamento por blocos (em s).

	Scanline Dinâmico		*Scanline* Estático	
	Lua	PVM	Lua	PVM
2 nós	1.347	0.545	2.898	0.931
4 nós	1.050	0.387	1.541	0.376
6 nós	0.935	0.433	1.181	0.426

Tabela 4.5: Tempos obtidos para o particionamento por *scanlines* (em s).

em apenas um processador, o visualizador, que recebia mensagens de todos os processadores calculadores. Para contornar este problema, adicionamos algumas cargas apenas aos processadores calculadores, de forma a fazer com que eles ficassem um pouco mais lentos e, deste modo, diminuissem a taxa de transmissão de mensagens ao processo visualizador. Neste caso, ambas as versões foram executadas com a adição destas cargas.

As figuras 4.8, 4.9, 4.10 e 4.11 mostram gráficos com a média dos tempos obtidos para cada particionamento (bloco dinâmico, bloco estático, *scanline* dinâmico e *scanline* estático) segundo a variação do número de processadores.

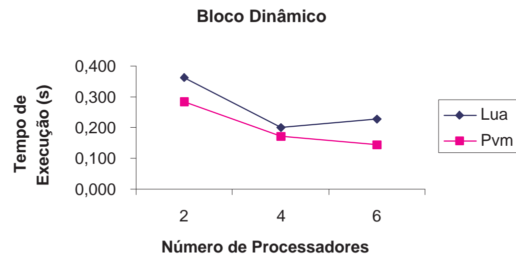


Figura 4.8: Tempos obtidos para o particionamento dinâmico por blocos.

Para esta aplicação, a versão PVM teve um desempenho superior em todos os casos observados. Uma diferença de comportamento observada é que no caso do particionamento por blocos, tanto no dinâmico quanto no estático, a versão dos Agentes Lua teve melhor desempenho quando utilizava apenas 4 processadores, enquanto a versão PVM teve um melhor desempenho quando utilizava 6 processadores. Porém, no caso do particionamento por *scanlines*, os melhores desempenhos ocorreram para 6 processadores na versão dos Agentes Lua e para apenas 4 processadores na versão PVM.

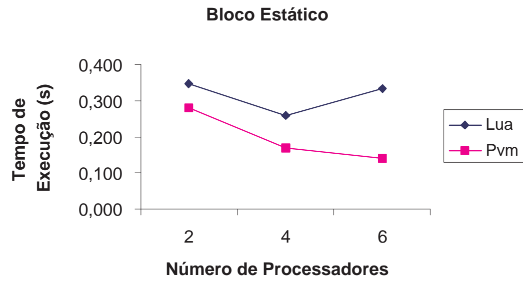


Figura 4.9: Tempos obtidos para o particionamento estático por blocos.

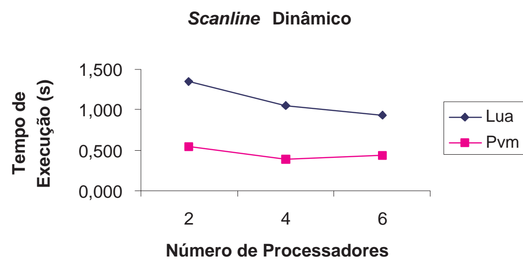


Figura 4.10: Tempos obtidos para o particionamento dinâmico por *scanlines*.

Para o particionamento por blocos, os Agentes Lua tiveram seu melhor desempenho com o particionamento dinâmico, utilizando 4 processadores, onde foram apenas 1,2 vezes mais lentos que PVM. O pior caso ocorreu para o particionamento estático com 6 processadores, onde os Agentes Lua foram 2,3 vezes mais lentos.

No caso do particionamento por *scanlines*, os Agentes Lua foram 2,2 vezes mais lentos que PVM, com o particionamento dinâmico com 6 processadores (melhor caso) e chegaram a ser 4,1 vezes mais lentos que PVM, com o particionamento estático com 4 processadores (pior caso observado em todos os resultados obtidos em ambas as aplicações).

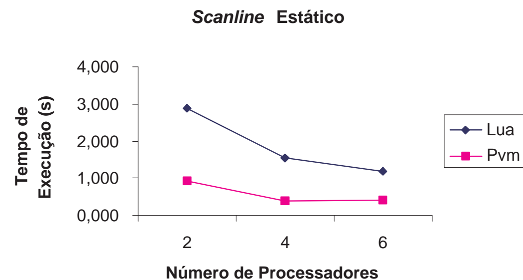


Figura 4.11: Tempos obtidos para o particionamento estático por *scanlines*.

4.3 Observações

Ainda é preciso solucionar algumas limitações do mecanismo dos Agentes Lua, como por exemplo, sua capacidade de armazenar de mensagens, que por ser limitada, faz com que ocorra perda de mensagens em algumas aplicações.

A solução destas limitações pode causar uma perda de desempenho um pouco maior deste mecanismo. No entanto, o resultado mais importante é que o mecanismo dos Agentes Lua é viável como mecanismo de comunicação, uma vez que os resultados obtidos mostram a mesma ordem de grandeza dos tempos observados em ambas as versões (Agentes Lua e PVM).

Capítulo 5

Considerações Finais

O objetivo inicial deste trabalho era fazer uma análise de desempenho do mecanismo de comunicação dos Agentes Lua. Para isso, começamos a estudar duas aplicações, das quais já tinhamos implementações em PVM.

Com base nas facilidades de uso que PVM oferece, surgiu a idéia de acrescentar algumas funcionalidades extras aos Agentes, tais como disparar novos agentes, empacotar dados para envio posterior e enviar uma mesma mensagem a mais de um agente. Este conjunto de recursos definiu uma nova camada de comunicação, que foi construída sobre o mecanismo básico dos Agentes.

Todos estes recursos, que foram inseridos no mecanismo dos Agentes Lua, tornaram-se bastante relevantes para a ferramenta, pois resolvem diversas limitações observadas anteriormente no uso do mecanismo.

Este conjunto de funcionalidades se mostrou razoavelmente completo pois, durante a implementação das duas aplicações, não foi necessária praticamente nenhuma alteração nesta nova camada. As alterações feitas, como por exemplo a introdução da extensão C, foram para resolver problemas de desempenho e não para acrescentar funcionalidade.

Com os dados obtidos pela análise de desempenho das duas aplicações, podemos notar que no pior caso, o mecanismo dos Agentes Lua foi quatro vezes mais lento que PVM, e no melhor caso o mecanismo dos Agentes Lua chega a ser três vezes mais rápido que PVM. Esta ordem de grandeza pode ser comparada com o uso de otimização em uma compilação. Há casos em que um mesmo programa chega a ser cinco vezes mais rápido quando compilado com opção de otimização.

Dado que Lua é uma linguagem interpretada, podemos dizer que mesmo com os resultados obtidos no pior caso, os Agentes Lua podem ser uma opção interessante para programação distribuída, devido a sua flexibilidade e também ao paradigma diferente do cliente/servidor tradicional que eles oferecem.

Cabe aqui lembrar que, quando medimos desempenho de uma aplicação paralela ou de uma ferramenta de programação, métricas diferentes devem ser levadas em consideração. A facilidade de construção de uma aplicação, assim como o tempo que o programador leva para implementá-la, pode em muitos casos ser mais importante que o tempo final de execução [Fos95]. Desta forma, o que se espera ganhar com este mecanismo é uma

grande flexibilidade para o programador, através das facilidades oferecidas pela linguagem utilizada e pelo mecanismo de comunicação proposto.

Esta flexibilidade pode ser particularmente interessante para o uso em prototipação. O uso de prototipação rápida é uma técnica que vem se tornando cada vez mais comum no desenvolvimento de programas [Bro95], principalmente no desenvolvimento de sistemas seqüenciais. O uso de prototipação também pode ser um recurso interessante para aplicações paralelas e distribuídas. Entretanto, apesar da variedade de linguagens existentes para o desenvolvimento de aplicações distribuídas, estas linguagens geralmente não oferecem os mecanismos de programação necessários para o suporte à prototipação. Os Agentes Lua trazem para o domínio de aplicações distribuídas todas as facilidades oferecidas por uma linguagem como Lua para prototipação rápida de sistemas.

Um tópico importante para este trabalho, que ainda deve ser resolvido, é implementar um mecanismo de controle de fluxo transparente para os Agentes Lua, de forma a tornar o mecanismo mais completo. Esta implementação resolveria o problema de ainda não termos nos Agentes Lua um buffer ilimitado. Na implementação atual, a mensagem recebida fica num buffer do sistema e só é consumida por um Agente quando este está livre para tratá-la. Para que os Agentes não tenham mais o problema de perda de mensagens, pode-se implementar um mecanismo para armazenar as mensagens recebidas num buffer interno, assim que elas chegam aos Agentes. Outra solução pode ser o uso de um protocolo mais confiável de transmissão de mensagens, como TCP no lugar de UDP. Neste sentido, podemos ainda ter duas versões dos Agentes Lua. Uma com um protocolo confiável, porém com uma perda de desempenho maior, e outra com um protocolo menos confiável, porém mais eficiente.

Outro problema que ainda deve ser resolvido é um problema de compatibilidade entre plataformas, que foi introduzido com o uso da extensão para C, descrita na seção 2.4.2. O problema acontece porque a representação de inteiros e de *floats*, principalmente, pode ser dependente da plataforma que se está utilizando. Neste caso, os dados podem ser empacotados em uma determinada ordem e ser lidos em outra, o que geraria erros nas mensagens. Cabe lembrar que este tratamento para a representação de dados irá comprometer mais um pouco o desempenho do mecanismo.

Além da correção destes problemas, novas funcionalidades poderiam ser incluídas aos Agentes. Na implementação das aplicações, pôde-se observar, com uma certa freqüência, a necessidade de sincronizar os Agentes. Algumas funções de sincronismo podem ser de grande utilidade para o desenvolvedor de aplicações baseadas nos Agentes Lua.

Outra funcionalidade que pode ser interessante é criar um *agente gerente* que possa monitorar os demais agentes de uma aplicação. Este gerente pode ter uma console ativa para que o usuário possa interagir com os demais Agentes, em tempo de execução. O usuário pode, desta forma, enviar informações de configuração, consultar o status de algum agente ou até mesmo alterar qualquer variável ou função Lua de um agente.

A solução dos problemas apontados e o acréscimo das funcionalidades descritas acima seriam uma importante contribuição para o desenvolvimento de um ambiente poderoso de programação distribuída baseada em Lua. Com este trabalho verificamos que tal ambiente é uma opção efetivamente viável para o desenvolvimento de aplicações paralelas distribuídas.

Apêndice A

Problemas Estudados

A.1 Problema dos N-Corpos

A.1.1 Trecho de Código do Agente Mestre

```
--     Parameter declarations and set defaults                    --
nparticles = 64                -- Total number of particles     --
                               -- default is 64                  --
nnodes = 4                     -- Number of node processes      --
                               -- default is 4                   --
MAX_ITERATIONS = 50            -- Total number of iterations    --
                               -- default is 50                  --
TOL = 1.0                      -- Tolerance                      --
                               -- default is 1.0                 --
TIMESTEP = 0.1                 -- Timestep                       --
                               -- default is 0.1                 --
SOFT = 0.025                   -- Softening parameters           --
                               -- default is 0.025               --
MEASURE_TIME = NO              -- need timing record?            --
                               -- default is NO                  --
HOW_OFTEN = 5                  -- how often?                     --
                               -- default is every 5 iters       --

infile = "input"               -- input filename                 --
init_file = "initialdist"      -- initial dist filemane          --

function main()
        -- master_tid          -- master task id                 --
        -- node_tids[MAXNODES] -- node task ids                  --
        -- inp                 -- input file descriptor          --
        -- nbody               -- initial particle list           --

  dofile(Directory .. infile)  -- read parameters                --

  com_spawn(nnodes)            -- Spawn node processes           --

  master_tid = com_mytid
end

function com_spawn_completed( hosts)
  node_tids = hosts
                               -- send initialization info       --
  com_initsend()
  com_pk("nparticles", nparticles)
```

```
  com_pk("nnodes", nnodes)
  com_pk("MAX_ITERATIONS", MAX_ITERATIONS)
  com_pk("TIMESTEP", TIMESTEP)
  com_pk("SOFT", SOFT)
  com_pk("TOL", TOL)
  com_pk("MEASURE_TIME", MEASURE_TIME)
  com_pk("HOW_OFTEN", HOW_OFTEN)
  com_pk("node_tids", node_tids)     -- send node task ids also --
  com_mcast(node_tids, "INITIALIZE") -- multicast message       --

  nbody=init_particles()
                              -- read initial distribution    --
  read_init_dist( nbody, nparticles, Directory .. init_file)

  com_pk("Directory", Directory)
  com_exc("dofile(Directory..'node.lua')")
  com_mcast(node_tids, "GET NODE CODE")

                              -- send particles to node 0     --
  send_particles(nbody, node_tids[1], 1, "INIT_DATA",
               "nparticles =recv_particles(nbody)")

  com_exc("node()")
  com_send(node_tids[1], "START EXECUTION")     -- only node 0  --
end

function master_end()
  print("The End...")

  com_exit_all()                      -- leave LUA Agent      --
  exit()
end

nodes_finished = 0

main()
```

A.1.2 Trecho de Código do Agente Simulador.

```
--      these are variables that describe the node configuration        --
--      They are used throught the program and remain unchanged         --
        -- node_tids                  -- tids of nodes                  --
        -- mytid                      -- my tid                         --
        -- dim                        -- dimension                      --
        -- master_tid                 -- tid of master                  --

--      input parameters                                                --
        -- nparticles                 -- total particles                --
        -- nnodes                     -- no. of nodes                   --
        -- MAX_ITERATIONS             -- no. iterations                 --
        -- TIMESTEP
        -- TOL
        -- SOFT
        -- MEASURE_TIME
        -- HOW_OFTEN

nbody = makenbody()

function node()
  if com_mytid == nil then
     print("Error while starting process. Aborting process...")
     exit()
  end
```

```
  mytid = com_mytid                     -- enroll in com                --
  com_initsend()
  hostname = node_tids[com_mytid]
  master_tid = com_myparent             -- get master tid               --
  dim = log2(nnodes)             -- dimension used in tree construction --
  mynum = mytid - 1
  root = makeroot()                     -- initialize tree structure    --
  mytree = root
                            -- distribute particles using ORB algorithm --
  mytree, nbody, myparticles = orb_decompose(mytree,nbody,nparticles)
  iteration = 0
  begin_node()
end

function begin_node()
  runaway_sync=0
  runaway_completed=0
  if iteration < MAX_ITERATIONS then
     form_tree(mytree, nbody, myparticles) -- construct local tree: ORB --
     if nnodes > 1 then                 -- exchange information among   --
        exchange_tree(root, iteration)  -- nodes: broadcast algo        --
     else
        resume_node()
     end
  else
     finish_node()
  end
end

function resume_node()
                             --  sendbody: runaway particle list        --
     compute_boundaries(root, dim)
     recompute_boundaries(root, dim)
     compute_forces(root, mytree, TIMESTEP, TOL, SOFT)
     sendbody = emptynbody()            -- pool particles               --
     myparticles, nbody, sendbody = collect_particles(mytree, nbody, sendbody)

     -- There is a need to synchronize the processes here so that       --
     -- runaway messages are not mixed with exchange ones.              --
     sync_to_runaway()
end

function node_runaway()
      local newparticles
                                        -- send runaway particles       --
      newparticles, nbody = runaway(root, sendbody, XMAX/100, nbody)
      myparticles = myparticles + newparticles
      cleanup(root, mytree, dim, mytid-1)      -- clean up              --
end

function node_runaway_received()
      iteration = iteration + 1
      begin_node()
end

function finish_node()
-- end of simulation. Send finish signal to master                      --
   signal_finish( master_tid)
end

function node_exit()
-- end of simulation. Send particle distribution to master              --
   send_dump_info( nbody, master_tid, mynum)
   com_exit()                           -- leave... done!               --
end
```

Referências Bibliográficas

[Ada79] Rationale for the design of ada programming language. *SIGPLAN Notices*, 14(6A, 6B), Junho 1979.

[AO93] G. Andrews e R. Olsson. *The SR Programming Language: Concurrency in Practice*. Benjamin Cummings, 1993.

[Bal90] Henri Bal. *Programming Distributed Systems*. Prentice-Hall, 1990.

[Bar96] Valmir C. Barbosa. *An Introduction to Distributed Algorithms*. The MIT Press, 1996.

[BH86] J. Barnes e P. Hut. A hierarchical o(n log n) force calculation algorithm. *Nature*, 324:446–449, 1986.

[Bic95] Lubomir F. Bic. Distributed computing using autonomous objects. Em *IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'95)*, Cheju Island, Korea, Agosto 1995.

[Bro95] F. P. Brooks. *The Mythical Man-Month: essays on software engineering*. Addison Wesley, anniversary edition, 1995.

[BST89] Henri E. Bal, Jennifer G. Steiner, e Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3), Setembro 1989.

[CG89] N. Carriero e D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[CK93] K. M. Chandy e C. Kesselman. CC++: A declarative concurrent object-oriented programming notation. Em *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.

[Cli81] William Douglas Clinger. Foundations of actors semantics. Relatório Técnico AI Technical Report 633, MIT, Maio 1981.

[CRI95] R. Cerqueira, N. Rodriguez, e R. Ierusalimschy. Uma experiência em programação distribuída dirigida por eventos. Em *PANEL95 — XXI Conferência Latino Americana de Informática*, páginas 225–236, Canela, 1995. SBC.

[CS93] D. E. Comer e D. L. Stevens. *Internetworking With TCP/IP Volume III: Client-server Programming and Applications BSD Socket Version.* Prentice-Hall, 1993.

[CT92] K. M. Chandy e S. Taylor. *An Introduction to Parallel Programming.* Jones and Bartlett Publishers, Boston, 1992.

[dBS97] Roberto de Beauclair Seixas. *Técnicas de Otimização em Visualização Volumétrica.* Tese de Doutorado, Dep. Informática, PUC-Rio, Rio de Janeiro, Brasil, 1997.

[FG93] M. Franklin e V. Govidan. The N-Body Problem: Distributed system load balancing and performance evaluation. Em *Proceedings of the 6th International Conference on Parallel and Distributed Computing Systems*, Louisville, KY, Outubro 1993. PDCS.

[Fos95] I. Foster. *Designing and Building Parallel Programs.* Addison Wesley, 1995.

[Gra95] Robert Gray. *Transportable Agents.* Tese de Doutorado, Department of Computer Science, Dartmouth College, 1995.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice – Hall, 1985.

[IFC95] R. Ierusalimschy, L. H. Figueiredo, e W. Celes. Reference manual of the programming language Lua version 2.1. Monografias em Ciência da Computação 08/95, PUC-Rio, Rio de Janeiro, Brazil, 1995. (available by ftp at `ftp.inf.puc-rio.br/pub/docs/techreports`).

[IFC96] R. Ierusalimschy, L. H. Figueiredo, e W. Celes. Lua—an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[Lov93] D. Loveman. High Performance Fortran. *IEEE Parallel and Distributed Technology*, 1(1):25–42, 1993.

[Mat87] F. Mattern. Algorithms for ditributed termination detection. *Distributed Computing*, 2:161–175, 1987.

[MRR96] S. Martins, C. Ribeiro, e N. Rodriguez. Ferramentas para programacao paralela em ambientes de memória distribuída. *Investigacion Operativa*, 5, 1996.

[Ous90] J. Ousterhout. Tcl: an embeddable command language. Em *Proc. of the Winter 1990 USENIX Conference.* USENIX Association, 1990.

[RR98] Isabel Rosseti e Noemi Rodriguez. Balanceamento de carga em uma implementação distribuída do problema dos n-corpos. Em *X SBAC-PAD*, Búzios, Brasil, Setembro 1998. (a ser publicado).

[RUIC96a] Noemi Rodriguez, Cristina Ururahy, Roberto Ierusalimschy, e Renato Cerqueira. The use of interpreted languages for implementing parallel algorithms on distributed systems. Em L. Bougé, P. Fraigniaud, A. Mignotte, e Y. Robert, editors, *Euro-Par'96 Parallel Processing — Second International Euro-Par Conference*, páginas 597–600, Volume I, Lyon, France, Agosto 1996. Springer-Verlag. (LNCS 1123).

[RUIC96b] Noemi Rodriguez, Cristina Ururahy, Roberto Ierusalimschy, e Renato Cerqueira. The use of interpreted languages for implementing parallel algorithms on distributed systems. Monografias em Ciência da Computação 13/96, PUC-Rio, Rio de Janeiro, Brazil, 1996.

[Ste90] W. Stevens. *UNIX Network Programming.* Prentice-Hall, 1990.

[Sun90] V. Sunderman. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, Dezembro 1990.

[Vel94] E. F. Velde. *Concurrent Scientific Computing.* Springer Verlag, New York, 1994.